Desarrollo de un compilador para un lenguaje funcional con gestión explícita de la memoria

Proyecto de Sistemas Informáticos

Autores: **Jesús Conesa, Ricardo López, Ángel Lozano**
Profesor director: **Clara María Segura Díaz**
Facultad de Informática
Universidad Complutense de Madrid
Julio 2006

Resumen

Este proyecto de la asignatura de Sistemas Infomáticos, titulado "Desarrollo del compilador de un lenguaje functional con gestión explícita de la memoria", está enmarcado en un proyecto más grande destinado al desarrollo de un lenguaje funcional de alto nivel, llamado SAFE, que permite que el programador gestione la memoria para que no se necesite un recolector de basura en tiempo de ejecución. Dispone además de un sistema de tipos que garantiza que no se producirán punteros descolgados en tiempo de ejecución. El proyecto se encarga de la implementación de la parte frontal del compilador, utilizando Alex y Happy como herramientas para el desarrollo del analizador léxico y sintáctico y Haskell para la implementación del analizador semántico y el renombramiento de identificadores, así como el resto de los módulos. Uno de estos módulos, que implementa una de las características del lenguaje y del compilador es el de las transformaciones amargativas, debido a que SAFE está concebido con dos sintaxis diferentes: una más cercana al programador —llamada dulce— que facilita la programación y otra más cercana a la implementación — llamada amarga— más apta para las fases posteriores del compilador. El objetivo de este proyecto es, pues, tener una herramienta para probar y validar la portabilidad de diferentes algoritmos desde Haskell hasta SAFE, permitiendo evaluar la viabilidad de este último lenguaje.

Abstract

This "Sistemas Informticos" project, entitled "Desarrollo del compilador de un lenguaje functional con gestin explcita de la memoria" — "Development of a compiler for a functional language with explicit memory management" — is part of a broader project devoted to the development of a high-level functional language, called SAFE, which allows the programmer to do memory management in order that a garbage collector would not be needed at runtime. Additionally, a type system guarantees that dangling pointers will not arise during execution. The project is aimed at the implementation of the compiler's front-end, using Alex and Happy as tools for the development of the lexical and syntactic parsers, and Haskell for the implementation of the semantic analyzer and the identifier renamer, as well as the rest of the modules. One of these modules, called desugarer, implements one feature of the language and its compiler: transforming from sweet to bitter syntax. This is because SAFE is conceived with two different syntaxes. The one nearer to the programmer —called sweet— that facilitates programming and the other nearer to the machine —called bitter— more amenable for the subsequent phases of the compiler. The aim of this project is, then, to have a tool to validate the portability of different algorithms from Haskell to SAFE, being also a test of the viability of this latter language.

Palabras clave

- **Compilador**: objetivo del proyecto.
- **Lenguaje funcional**: característica principal del lenguaje desarrollado.
- **Gestión de Memoria**: el lenguaje permite al programador gestionar la memoria de forma explícita.
- **Transformaciones**: aplicamos transformaciones a la sintaxis del lenguaje.

Índice general

1. Introducción **7**
1.1. Contexto del proyecto . . . 7
1.2. Objetivos del proyecto . . . 8
1.3. Plan de trabajo . . . 8

2. El lenguaje SAFE **11**
2.1. Conceptos generales del lenguaje . . . 11
2.2. Sintaxis *amarga* . . . 13
2.3. Sintaxis *dulce* . . . 15

3. El compilador de SAFE **17**
3.1. Fases del compilador . . . 17
3.2. Herramientas utilizadas . . . 18
3.2.1. Haskell/GHC . . . 18
3.2.2. Alex . . . 19
3.2.3. Happy . . . 20
3.2.4. *Pretty Printer* . . . 22

4. Fase de análisis léxico **25**
4.1. Unidades léxicas . . . 25
4.2. Analizador léxico . . . 27
4.3. Postproceso . . . 29
4.3.1. Inserción de los delimitadores de ámbito . . . 29
4.3.2. Distinción de las categorías léxicas reconocidas como iguales . . . 30
4.3.3. Extracción de la expresión principal . . . 31

5. Fase de análisis sintáctico **33**
5.1. El árbol abstracto . . . 33
5.2. Impresión amigable del árbol . . . 35
5.3. Analizador sintáctico . . . 37
5.4. Postproceso . . . 40

6. Fase de comprobación semántica **41**
6.1. Comprobaciones semánticas . . . 41
6.2. Ámbito de los identificadores . . . 42
6.3. Analizador semántico . . . 42

7. Transformaciones de *desazucaramiento* **47**
7.1. Transformaciones simples 47
7.2. Transformaciones complejas 49
7.2.1. Regla vacía 49
7.2.2. Regla cuando todos los patrones son una variable 49
7.2.3. Regla cuando todos los patrones son constructoras 50
7.2.4. Regla mixta 50
7.2.5. Ejemplo 51

8. Ejemplo completo **53**

9. Conclusiones **57**

A. Código del análisis léxico **59**

B. Código del análisis sintáctico **61**

C. Código del análisis semántico **63**

D. Código de las transformaciones **65**

Capítulo 1

Introducción

1.1. Contexto del proyecto

La mayoría de los lenguajes funcionales permiten que el programador se olvide de los detalles de manejo de la memoria. El sistema en tiempo de ejecución se encarga de reservar memoria a medida que se van evaluando las expresiones del programa mientras exista memoria disponible suficiente. En caso de que se agote la memoria comienza a ejecutarse un recolector de basura que determina qué partes de la memoria están muertas y por tanto pueden ser reutilizadas. Esto implica normalmente la suspensión de la ejecución del programa original. En caso de que el recolector haya logrado recuperar suficiente cantidad de memoria se puede continuar con la ejecución, mientras que en caso contrario el programa aborta. Este modelo es aceptable en muchas situaciones puesto que los programas no se ven oscurecidos por detalles de bajo nivel sobre el manejo de la memoria. Pero en otros contextos puede no serlo por los siguientes motivos:

- El retraso introducido en la ejecución por la recolección de basura impide proporcionar la respuesta en un determinado tiempo de reacción, como se requiere en programas con respuesta en tiempo real o altamente interactivos.
- El fallo de los programas debido al agotamiento de la memoria puede provocar daños personales o económicos inaceptables a los usuarios de los programas. Este es el caso de aplicaciones críticas en seguridad.

Por otra parte, muchos lenguajes imperativos ofrecen mecanismos de bajo nivel para reservar y liberar memoria que el programador puede utilizar para crear y destruir dinámicamente estructuras de datos utilizando punteros. Estos mecanismos proporcionan al programador un control completo sobre el uso de la memoria pero son fuente de numerosos errores. Algunos problemas bien conocidos son los punteros descolgados, la compartición indeseada con efectos laterales inesperados, y la saturación de la memoria con basura.

Ni el enfoque implícito de los lenguajes funcionales ni el explícito de los imperativos proporciona una lógica clara con la que el programador pueda razonar sobre el uso de la memoria. En particular, es difícil anticipar, o demostrar, una cota superior de la cantidad de memoria que garantice la ausencia de fallos en la ejecución debidos al agotamiento de la memoria.

En el Departamento de Sistemas Informáticos y Programación de la Universidad Complutense de Madrid, Clara Segura participa en un proyecto coordinado con las Universidades de Castilla La Mancha, Politécnica de Valencia y Málaga (TIN2004-07943-C04) y dirigido por Ricardo Peña. En el seno de este proyecto, el grupo de la UCM trabaja en el área del *Código con Demostración Asociada* (en inglés *Proof Carrying Code* o PCC). En este contexto han desarrollado un enfoque semiexplícito

del manejo de la memoria [PS04] definiendo un lenguaje funcional llamado *SAFE* en el que el programador coopera de forma moderada con el sistema de manejo de la memoria proporcionando cierta información sobre el uso que se pretende hacer de las estructuras de datos. Por ejemplo, puede indicar que una estructura de datos particular no se va a necesitar más y que consecuentemente se puede destruir de forma segura y reutilizar la memoria que ocupa. También permite el control de la compartición entre distintas estructuras de datos. La gestión de la memoria se realiza mediante regiones. Los beneficios de este enfoque son fundamentalmente que no es necesario un recolector de basura y que es posible razonar más fácilmente sobre la cantidad de memoria que consume un programa.

Pero aun más importante que las facilidades concretas que se le ofrecen al programador, es la definición de un sistema de tipos que garantiza que el uso de las mismas se hace de forma segura [PS04]. Este sistema de tipos garantiza que no se crean punteros descolgados en la memoria. Se puede demostrar correcto con respecto a la semántica operacional del lenguaje, definida formalmente en [PSM06]. La definición del sistema de tipos incluye un análisis de compartición definido mediante interpretación abstracta [PSM06] que permite determinar qué variables están en peligro por el hecho de destruir una estructura de datos.

1.2. Objetivos del proyecto

El objetivo de este proyecto es implementar una herramienta que constituya la parte frontal de un compilador para *SAFE*, con las características mencionadas en el apartado anterior. Dicha herramienta servirá para contemplar la utilidad y los beneficios del mismo, a base de ejecutar diversos ejemplos de programas implementados en dicho lenguaje, con el fin de motivar el desarrollo completo de un compilador con estas características.

Para la implementación del mismo, hemos distinguido una serie de fases que describiremos en el siguiente apartado. De todas ellas nos hemos interesado en particular por el analizador léxico, el analizador sintáctico, la comprobación y renombramiento de identificadores (o analizador semántico), y las transformaciones *amargativas*, que son las que implementaremos y describiremos en detalle a lo largo de este trabajo. El resto de las fases serán proporcionadas por colaboradores del proyecto.

En la figura 3.1 citamos cada una de ellas, indicando tanto el orden que deben seguir en el proceso de compilación como los tipos de datos de entrada que procesan y los resultados que producen.

1.3. Plan de trabajo

Como se ha mencionado anteriormente, este proyecto se ha centrado en algunas etapas del compilador, para cada una de las cuales se ha dedicado un capítulo completo que se ocupa de describir tanto el funcionamiento como la implementación de la misma. A continuación daremos una descripción de cada uno de ellos:

- En el capítulo 3 dedicaremos una sección para dar una descripción detallada de cada una de las fases del proyecto, y además incluiremos otra sección en la cual mencionaremos las herramientas utilizadas para el desarrollo del proyecto, explicando detalladamente el uso de las mismas. En particular dedicaremos un apartado para la herramienta Alex, otro para Happy y por último uno para el compilador de Haskell GHC *(The Glasgow Haskell Compiler)*.

- En el capítulo 4 nos centraremos en el analizador léxico. En particular, dedicaremos una sección para especificar las unidades léxicas del lenguaje SAFE, y otra para explicar el funcionamiento de la implementación del mismo y cómo contribuye la herramienta Alex en la misma, incluyendo aquellas secciones de código más relevantes junto con sus correspondientes explicaciones. Finalmente se dedicará otro apartado al postproceso, es decir, detallaremos el proceso que se llevará a cabo con el resultado obtenido antes de proporcionarlo a la siguiente fase. Concretamente se hará mención a la regla del formato y a la estructura básica de un programa implementado en este lenguaje.

- El capítulo 5 lo dedicaremos al analizador sintáctico. Concretamente, habrá una sección dedicada al árbol abstracto, junto con una especificación completa del mismo, que es el tipo de datos que manejaremos durante el resto de las fases que siguen a ésta. También explicaremos el funcionamiento de esta implementación y el modo de empleo de la herramienta Happy, incluyendo nuevamente código relevante y explicaciones. Como antes, dedicaremos otro apartado al postproceso, que se ocupa de resolver algunas dificultades encontradas durante el análisis sintáctico.

- El capítulo 6 se dedica a la fase de comprobaciones semánticas. En primer lugar comentaremos informalmente una lista de restricciones que debe cumplir un programa implementado en el lenguaje SAFE, y posteriormente especificaremos formalmente una serie de reglas semánticas para las definiciones, los patrones y las expresiones, así como unas reglas de ámbito para los identificadores. Incluiremos además una breve descripción de la implementación del analizador semántico.

- En el capítulo 7, dedicado a las transformaciones de *desazucaramiento*, se especifica una lista de reglas formales para convertir la sintaxis dulce en sintaxis amarga. El hecho de que hayamos dedicado un capítulo a esta etapa es debido a la complejidad de dichas reglas, por ello merecen una explicación detallada.

- Finalmente, en el capítulo 9 trataremos las conclusiones obtenidas del proyecto.

Además de éstos, se incluiye un capítulo 8 adicional con un ejemplo completo que ilustra el resultado del proceso realizado en cada fase. También se incluýe una serie de apéndices, referenciados desde los capítulos, donde se adjuntan listados de código de la implementación.

Capítulo 2

El lenguaje SAFE

2.1. Conceptos generales del lenguaje

Una forma posible de presentar el lenguaje SAFE sería sencillamente "como un hecho": esta es la sintaxis, esta es la semántica y estos son los programas que pueden construirse con ella. Entonces, el lector juzgaría sus ventajas y sus limitaciones. Este enfoque es el que seguiremos posteriormente, pero antes de embarcarnos en los detalles, daremos a conocer el razonamiento que ha llevado al diseño del lenguaje, en forma de un conjunto de definiciones y axiomas.

Definición 1 *Una* **región** *es un área de memoria contigua del montón donde pueden construirse, leerse, o destruirse estructuras de datos (ED). Tiene un tamaño conocido en tiempo de ejecución y se asigna y libera como un todo, en tiempo constante.*

Definición 2 *Una* **celda** *es un espacio pequeño de memoria, lo bastante grande como para albergar un constructor de datos. En términos de implementación, una celda contiene la marca (o el puntero al código) del constructor, y una representación de las variables libres a las que se aplica el constructor. Éstas pueden consistir bien en valores básicos, o bien en punteros a valores no básicos.*

Por "suficientemente grande" se quiere decir que una celda liberada puede ser reutilizada directamente por el sistema. Una implementación sencilla podría tener en una celda el espacio suficiente para almacenar el constructor más grande. Una aproximación más eficiente podría establecer una lista de tamaños fijos de celdas, todos ellos múltiplos del tamaño mínimo. En este apartado no entraremos en detalles más concretos sobre este tema. La idea importante es que las celdas puedan ser reutilizadas inmediatamente en un tiempo de ejecución constante.

Definición 3 *Una* **estructura de datos***, en lo que sigue ED, es el conjunto de celdas obtenidas empezando desde una celda considerada como raíz, y tomando el cierre transitivo de la relación $C_1 \rightarrow C_2$, donde C_1 y C_2 son celdas del mismo tipo T, y en C_1 hay un puntero a C_2.*

Ello quiere decir que, por ejemplo en una lista de tipo `[[a]]`, estamos considerando como una ED todas las celdas pertenecientes a la lista más externa, pero no aquellas que pertenecen a las listas individuales internas. Cada una de estas últimas constituyen una ED aparte.

Axioma 1 *Una ED reside completamente en una región.*

El razonamiento que hay detrás de esta decisión es que, cuando una región muere, todas las estructuras de esa región mueren también. Si partiéramos una ED en varias regiones, entonces obtendríamos estructuras de datos parcialmente destruidas. Este axioma impone una obligación en las estructuras de datos: sus posiciones recursivas solo deberían admitir parámetros pertenecientes a la misma región, y la celda resultante debe construirse también en esa misma región.

Axioma 2 *Dadas dos ED, una de ellas puede ser parte de la otra, o pueden compartir una tercera ED.*

Aplicando el Axioma 1, las tres deben residir en una sola región. Se desea tener en el lenguaje tanta compartición como sea posible, ya que es la clave para conseguir eficiencia. En un caso extremo, se podría recuperar la programación funcional convencional teniendo una sola región en todo el programa, y una compartición potencial entre todas las EDs.

Axioma 3 *Los valores básicos —enteros, booleanos, etc.— no reservan celdas dentro de las regiones, sino que viven dentro de las celdas de las EDs, o en la pila.*

Axioma 4 *Una función de n parámetros puede acceder, como mucho, a $n + 2$ regiones.*

- *Puede acceder a las EDs de sus n parámetros, y cada una puede residir en una región diferente.*
- *Toda función que construye una ED como resultado debe construirla en su* **región de salida**. *Hay como mucho una región de salida por función. La entrega del identificador de esta región es responsabilidad del invocador. La función recibe la región de salida como un parámetro adicional.*
- *Toda función tiene una* **región de trabajo** *donde puede crear EDs intermedias. La región de trabajo tiene el mismo tiempo de vida que la función: se reserva en cada invocación y se libera cuando esta termina.*

Las funciones dejan su resultado en una región de salida perteneciente al invocador para que los resultados intermedios computados por la función puedan borrarse con seguridad cuando la función termine. De este modo, las regiones de trabajo del montón pueden apilarse según se va invocando a funciones de forma anidada, y ello facilita razonar acerca del tamaño máximo del montón que el programa necesita.

Axioma 5 *Si un parámetro de una función es una ED, puede ser destruida por la función. Se permite a las funciones comportarse de este modo si el programador así lo decide. Se dice que el parámetro está* **condenado** *porque la destrucción depende de la definición de la función, no de su uso. Ello quiere decir que todo uso de ciertas funciones destruirán ciertos parámetros. La destrucción implica que el invocador puede suponer con seguridad que todas las celdas ocupadas por la ED condenada serán recuperadas y reutilizadas por el sistema en tiempo de ejecución.*

Axioma 6 *Las capacidades que una función tiene en sus EDs accesibles y regiones son las siguientes:*

- *Una función sólo puede leer de las ED que sean parámetros de sólo lectura.*
- *Una función puede leer (antes de destruirla), y debe destruir, una ED que sea un parámetro condenado.*

prog	→	$dec_1; \ldots; dec_n; expr$	
dec	→	$f\ \overline{x_i}^n\ r = expr$	{función polimórfica, recursiva simple}
		$\mid f\ \overline{x_i}^n = expr$	
expr	→	a	{átomo: literal c o variable x}
		$\mid x@r$	{copia}
		$\mid x!$	{reuso}
		$\mid (f\ \overline{a_i}^n)@r$	{función de aplicación, que construye algo}
		$\mid (f\ \overline{a_i}^n)$	{función de aplicación}
		$\mid (C\ \overline{a_i}^n)@r$	{constructor de aplicación}
		$\mid$ **let** $x_1 = expr_1$ **in** $expr$	{no recursivo, monomórfico}
		$\mid$ **case** x **of** $\overline{alt_i}^n$	{case de sólo lectura}
		$\mid$ **case!** x **of** $\overline{alt_i}^n$	{case destructivo}
alt	→	$C\ \overline{x_i}^n \rightarrow expr$	

Figura 2.1: Sintaxis amarga del lenguaje *Safe*

- *Una función puede construir, leer, o destruir EDs, bien en su región de trabajo o bien en su región de salida.*

Tener parámetros destruibles es una forma simple de decirle al sistema que una ED ya no se va a usar. La propiedad de que la destrucción sea segura puede comprobarse mediante un sistema de tipos, los parámetros condenados pueden ser reflejados en el tipo de la función de modo que los usuarios conozcan dicha característica mirando simplemente el tipo.

2.2. Sintaxis *amarga*

La sintaxis de SAFE puede verse en la figura 2.1. Se trata de un lenguaje funcional de primer orden ímpaciente donde la compartición se impone usando variables en las llamadas a las funciones y en la aplicación de constructores. Esta sintaxis define el lenguaje núcleo que resulta de la compilación de un lenguaje de mayor nivel similar a Haskell.

Un programa en SAFE, *prog*, es una secuencia de definiciones de funciones polimórficas recursivas simples seguidas por una expresión principal *expr* cuyo valor es el resultado del programa. Cada definición de función sólo puede llamar a una función previamente definida y la expresión principal puede llamar a cualquiera de ellas. Aquellas funciones que construyan una ED tendrán un parámetro adicional r, la región de salida, donde se construirá la ED resultante. En el lado derecho de la expresión solo se puede usar r y su propia región de trabajo, *self*. También se permiten definiciones de tipos de datos algebraicos polimórficos. Se definen mediante declaraciones **data**. En todos los ejemplos que damos, usaremos listas $[a]$ con constructores $[\,]$ y $(:)$, y árboles binarios *Tree* a con los constructores *Empty* y *Node*. Las expresiones del programa incluyen lo habitual: variables, literales, funciones y constructores de aplicaciones, y también expresiones **let** y **case**, sin embargo hay algunas expresiones adicionales que explicaremos cuidadosamente.

Si x es una ED, la expresión $x@r$ representa una copia de la ED accedida desde x, cuando x no resida en r. Por ejemplo, si x es una lista residente en $r' \neq r$, entonces $x@r$ es una nueva lista con la misma estructura que x, compartiendo sus elementos con x.

La expresión $x!$ representa la reutilización de la ED destruible a la que x apunta. Esto es útil cuando no se quiere destruir completamente un parámetro condenado sino reutilizar parte de él. Podríamos hacer una copia permitiendo que x sea destruida pero sería menos eficiente.

En las aplicaciónes de función tenemos una sintaxis especial $@r$ para incluir el parámetro adicional de la región de salida. Utilizando la misma sintaxis, expresamos que un constructor de aplicación debe residir en la región r.

La expresión **case!** Indica que el constructor externo de x se libera después del encaje de patrones de modo que x ya no sea accesible. Las subestructuras recursivas pueden destruirse explícitamente

```
reverseF :: ∀a, ρ1, ρ2.[a]@ρ1 → ρ2 → [a]@ρ2
reverseF xs r = (revauxF xs [ ]@r)@r
revauxF xs ys r = case xs of
                      [ ] → ys
                      x : xx → (revauxF xx (x : ys)@r)@r

reverseD :: ∀a, ρ1, ρ2.[a]!@ρ1 → ρ2 → [a]@ρ2
reverseD xs r = (revauxD xs [ ]@r)@r
revauxD xs ys r = case! xs of
                      [ ] → ys
                      x : xx → (revauxD xx (x : ys)@r)@r
```

Figura 2.2: Versión funcional y destructiva de inversión de listas

```
insertD :: ∀a, ρ.a → Tree a!@ρ → ρ → Tree a@ρ
insertD x t r = case! t of
                    Empty → (Node Empty@r x Empty@r)@r
                    Node i y d → let c = compare x y
                                 in case c of
                                        LT → (Node (insertD x i)@r y d!)@r
                                        EQ → (Node i! y d!)@r
                                        GT → (Node i! y (insertD x d)@r)@r
```

Figura 2.3: Versión destructiva de inserción en árbol de búsqueda binario

en el código correspondiente mediante otro **case!** o reutilizada mediante $y!$. Una variable condenada puede leerse pero, una vez que sea destruida o se reutilice su contenido en otra estructura, no puede accederse a ella otra vez.

A continuación mostraremos con varios ejemplos cómo utilizar las facilidades del lenguaje. En algunos de ellos escribiremos $x!$ o $(C\ \overline{a_i}^n)@r$ en los parámetros de las aplicaciones para abreviar (habría que usar en su lugar una asignación **let**).

El primer ejemplo es la función que invierte una lista. En la figura 2.2 mostramos dos versiones de dicha función. En ambas usamos la función auxiliar habitual con un parámetro acumulador. La primera versión, *reverseF*, es la versión funcional en la que se conserva la lista original. Nótese que la única diferencia con respecto a la función que un programador funcional escribiría es la región de salida r. El parámetro acumulador es construido en la región de salida de modo que al final contenga el resultado de la función. La segunda versión, *reverseD*, es una versión destructiva en la que se pierde la lista original. Nótese que la única diferencia con la versión anterior es que usamos **case!** sobre la lista original. La aplicación recursiva de la función la destruye completamente. Quien invoque a *reverseD* debería saber que el argumento se pierde en el proceso de inversión, y no deberían intentar utilizarla más.

El siguiente ejemplo ilustra la reutilización de una estructura condenada. Es la función que inserta un elemento en un árbol binario de búsqueda. En esta ocasión únicamente mostraremos, en la figura 2.3, la versión destructiva, en la cual el árbol original se destruye parcialmente. Todo el árbol salvo la ruta desde la raíz hasta el elemento insertado se reutiliza para construir el nuevo árbol pero esas partes no pueden accederse más desde el árbol original. Nótese que cuando el elemento insertado ya está en el árbol (caso EQ), tenemos que reconstruir el árbol que acabamos de destruir. La versión funcional se obtiene quitando los símbolos ! y devolviendo t en el caso EQ, como si no se hubiera destruido.

Los tipos incluidos en las definiciones de función son los que el compilador infiere en la fase de inferencia de tipos Hindley- Milner.

prog	→	$def_1; \ldots; def_n; exp$	
def	→	*defFun*	{ecuación de definición de función, todas juntas}
		| **data constr** {**id**}* @ {**id**}$^+$ = $altDato_1; \ldots; altDato_n$	{definición de tipo de datos}
		| **id** :: *expTipo*	{tipo de una función, debe ir antes de las definiciones}
altDato	→	**constr** @ **id**	{constructor de alternativa sin parámetros}
		| (**constr** {*expTipo*}$^+$) @ **id**	{constructor de alternativa con parámetros}
		| (*expTipo* **cinfija** *expTipo*) @ **id**	{constructor de alternativa infijo}
expTipo	→	**id**	
		| **constr**	{constructor de tipo sin parámetros}
		| **constr** ({*expTipo*}$^+$) [!] @ {**id**}$^+$	{constructor de tipo con parámetros}
		| (*expTipo* {, *expTipo*}$^+$) [!] @ **id**	{constructor de tipo para tuplas}
		| [*expTipo*] [!] @ **id**	{constructor de tipo para listas}
		| *expTipo* → *expTipo*	{separador de parámetros}
		| (*expTipo*)	
defFun	→	(*pat* | **id** {*pat*[!]}* | *pat*[!] **op** *pat*[!]) *der*	
der	→	= *exp* [**where** {*def*{; *def*}*}]	
		| {| *exp* = *exp*}$^+$ [**where** {*def*{; *def*}*}]	
pat	→	*lit*	
		| **id**	
		| [ ]	
		| (*pat* {, *pat*}$^+$)	
		| *pat* **cinfija** *pat*	
		| **constr** {*pat*}*	
		| (*pat*)	
exp	→	*a*	{átomo: literal c o variable x}
		| x@r	{copia}
		| x!	{reutilización}
		| $(f\ \overline{exp_i}^n)$@r	{aplicación de función, que construye algo}
		| $(f\ \overline{exp_i}^n)$	{aplicación de función}
		| $(C\ \overline{exp_i}^n)$@r	{aplicación de constructor}
		| *exp* **cinfija** *exp*	{aplicación de constructor (infijo)}
		| *exp* **op** *exp*	{aplicación de operador infijo}
		| **let** $\overline{def_i}^n$ **in** *exp*	{recursivo, monomórfico}
		| **case** *exp* **of** $\overline{alt_i}^n$	{case de sólo lectura}
		| **case!** *exp* **of** $\overline{alt_i}^n$	{case destructivo}
		| (*exp*)	{expresión parentizada}
alt	→	*pat* → *expr*	

Figura 2.4: Sintaxis dulce del lenguaje *Safe*

2.3. Sintaxis *dulce*

Como se ha indicado anteriormente, la sintaxis *amarga* pretende ser el núcleo de un lenguaje de mayor nivel similar a Haskell. La encargada de soportar este lenguaje de alto nivel es la sintaxis *dulce*. Si sólo se contara con la sintaxis *amarga* como medio de programación, la programación en SAFE resultaría muy tediosa. Por ello, es necesario una sintaxis que ofrezca más facilidades al programador, la sintaxis *dulce*, con el fin de simplificar el trabajo al mismo. A la hora de programar el usuario debe ceñirse a ésta. Posteriormente, mediante las transformaciones de *desazucaramiento* —descritas en el capítulo 7—, los programas escritos a partir de la sintaxis *dulce* se convertirán en su equivalente *amarga*.

Esta gramática no será todavía la utilizada por el analizador sintáctico porque es ambigua, pero muestra de forma compacta el lenguaje que se quiere definir.

Las reglas que la definen se muestran en la figura 2.4. A continuación se detallan sus principales ventajas.

En la sintaxis *amarga* únicamente son posibles las definiciones de función, sin embargo en la sintaxis *dulce* se permite la definición de declaraciones de tipo y de datos. Las primeras serán de gran utilidad para especificar los tipos de los datos empleados así como el de las funciones. Las segundas ofrecerán al programador la posibilidad de crear sus propias estructuras para almacenar información. Ambas se sustentan en las expresiones de tipo, `expTipo`, para poder declararse. Dichas

```
reverseD :: [a]!@ρ1 → ρ2 → [a]@ρ2
reverseD xs r = (revauxD xs [ ]@r)@r
revauxD :: [a]!@ρ1 → [a]@ρ2 → ρ2 → [a]@ρ2
revauxD [ ]! ys r = ys
revauxD (x : xx)! ys r = (revauxD xx (x : ys)@r)@r
```

Figura 2.5: Versión destructiva de inversión de listas (sintaxis *dulce*)

```
insertD :: a → Tree a!@ρ → ρ → Tree a@ρ
insertD x Empty! r = (Node Empty@r x Empty@r)@r
insertD x (Node i y d)! r
  | x < y = (Node (insertD x i)@r y d!)@r
  | x = y = (Node i! y d!)@r
  | x > y = (Node i! y (insertD x d)@r)@r
```

Figura 2.6: Versión destructiva de inserción en árbol de búsqueda binario

expresiones se han jerarquizado en `expTipo`, `expTipob` y `expTipoc`.

En la sintaxis dulce se ofrece la posibilidad de definir una función a partir de varias declaraciones de función —separando cada caso en una ecuación diferente—. Sin embargo, en la sintaxis amarga deben reunirse en una sola declaración todos los casos, de ahí que la parte derecha de una función en sintaxis amarga sea habitualmente un **case**.

En la figura 2.6 se observa que la función *insertD* está definida en dos declaraciones de función, la segunda de las cuales se sirve de guardas para discernir casos. Sin embargo, en la versión amarga, mostrada en la figura 2.3, la función *insertD* se define mediante una única declaración que tiene como parte derecha un **case** cuyas alternativas se anidan a su vez en otro **case**.

Las declaraciones de función pueden definirse de muchas formas. La parte izquierda puede ser un patrón, un identificador seguido de patrones o incluso un operador rodeado de patrones. La parte derecha puede ser una expresión simple o una lista de guardas.

Otra facilidad de la sintaxis dulce es la posibilidad de aadir cláusulas **let**. Mediante éstas se pueden formar expresiones con variables definidas localmente a la propia expresión. Los **let** sólo pueden contener declaraciones de función, nunca declaraciones de datos. Las funciones también pueden tener opcionalmente definiciones locales. Mediante la cláusula **where** se puede declarar expresiones al final de una función que sólo sean visibles dentro de la misma y así poder reutilizarlas. Al igual que sucede con los **let**, no es posible declarar definiciones de datos dentro de los **where**.

Capítulo 3

El compilador de SAFE

3.1. Fases del compilador

Como puede verse en la figura 3.1, la primera fase la constituye el analizador léxico, cuyo parámetro de entrada es una lista de caracteres, correspondiente al programa fuente que el usuario desea compilar. La unidad más pequeña que maneja este analizador es el carácter.

El objetivo es obtener una lista de **unidades léxicas** (o **tokens**), en lo que sigue UL, que será la unidad utilizada en la siguiente etapa: el analizador sintáctico. Dichas ULs pueden ser identificadores, operadores infijos, palabras reservadas, símbolos especiales del lenguaje, etc. Para ello se implementa un autómata finito de estados que reconozca cada una de las ULs definidas en el mismo. Los detalles acerca de la implementación, así como la especificación de las ULs pueden encontrarse en el capítulo 4.

El siguiente paso consiste en analizar la lista de ULs obtenida en la fase anterior y construir a partir de la misma el **árbol abstracto** que representa al programa correspondiente a la sintaxis dulce especificada en el capítulo 2. La definición del árbol puede encontrarse en el capítulo 5. Para implementar esta fase hemos definido una gramática incontextual, jerarquizada adecuadamente para explicitar tanto la asociatividad y como la prioridad de los operadores definidos en la fase anterior, y así evitar cualquier posible ambigüedad. Los detalles de dicha implementación los daremos en el capítulo 5.

Una vez construido el árbol abstracto, es necesario comprobar que la sintaxis cumple una serie de restricciones semánticas, por ello incluimos una fase de comprobación semántica en la que además realizaremos un proceso de renombramiento de los identificadores que se definen en diferentes ámbitos, para asegurarnos de que a cada identificador le corresponde un nombre único. En esta ocasión, y solo en caso de que el programa cumpla las restricciones impuestas, el resultado será el mismo árbol que recibió de la etapa anterior, salvo renombramientos de identificadores. Explicaremos las reglas semánticas y el ámbito de identificadores en el capítulo 6.

La siguiente fase, habitual en los compiladores de lenguajes funcionales, consiste en inferir los tipos de las expresiones del programa, y realizar las comprobaciones de tipo pertinentes. El algoritmo de inferencia será una modificación del algoritmo *Hindley-Milner*. La implementación de esta etapa ha sido llevada a cabo por colaboradores, por lo que no entraremos en detalles ni dedicaremos un capítulo al mismo. El resultado será el mismo árbol abstracto de antes pero *decorado* con la información de tipos inferida en esta fase.

A continuación transformaremos el programa, escrito con sintaxis dulce, al lenguaje núcleo (sintaxis amarga) que explicamos en el capítulo anterior. Esta fase es necesaria para simplificar el proceso que realizarán las dos fases siguientes, a saber, el análisis de compartición y la inferencia segura de tipos. Para ello definiremos una serie de reglas de transformaciones amargativas, detalladas

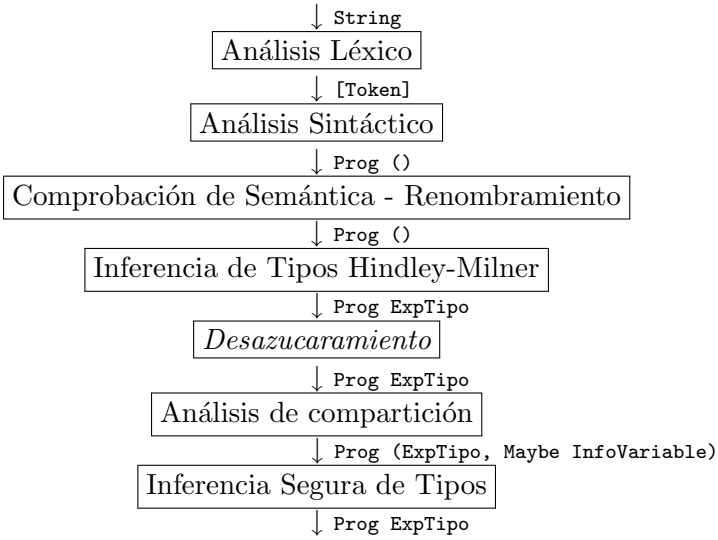

Figura 3.1: Fases del compilador

en el capítulo 7.

Aunque las últimas fases no han sido implementadas en este proyecto, dedicaremos unas líneas a aclarar la necesidad de las mismas. En primer lugar, el análisis de compartición es necesario para decorar el programa con información de compartición, localización y destrucción de estructuras de datos, de modo que podamos prescindir de un recolector de basura. Y por último, la inferencia segura de tipos es la que asigna los tipos finales mostrados en las figuras 2.2 y 2.3. El sistema de tipos garantiza que si un programa pasa esta fase, los errores comunes de los programadores, tales como las referencias a memoria no válidas o basura que no se libera correctamente, no ocurran en tiempo de ejecución.

3.2. Herramientas utilizadas

3.2.1. Haskell/GHC

Como lenguaje principal en el cual están basadas el resto de las herramientas se ha elegido Haskell, un lenguaje funcional perezoso polimórfico de orden superior. Se ha dicho así por la forma natural en la que los lenguajes funcionales atacan algunos de los problemas derivados de las diferentes fases del compilador y la potencia inherente a este tipo de lenguajes, aunque también venía impuesto por el marco del proyecto. Por otro lado la elección de GHC (The Glasgow Haskell Compiler), un compilador y no un intérprete como otras implementaciones de Haskell tales como *HUGS*, vino dada por la envergadura del proyecto y la necesidad de tener módulos compilados para la interacción de las diferentes herramientas. Más información al respecto, así como instrucciones de compilación y posibilidades del lenguaje en `http://www.haskell.org/ghc/` y `http://www.haskell.org/haskellwiki/Haskell`. También otra referencia importante es Hoogle (`http://www.haskell.org/hoogle/`), una herramienta para buscar cualquier definición de Haskell y su módulo asociado.

3.2.2. Alex

Alex es una herramienta para generar analizadores léxicos en Haskell. Dada una descripción de las unidades léxicas a reconocer en forma de expresiones regulares Alex, genera un módulo Haskell que contiene código para analizar texto efectivamente. Es muy parecido a las herramientas `lex` y `flex` para C/C++.

Una especificación léxica en Alex se guarda normalmente en un fichero con una extensión .x . La forma general de un fichero Alex es la siguiente:

```
alex := [ código ] [ wrapper ] { macros } token ':-' { regla } [ código ]
```

El archivo comienza y termina con fragmentos opcionales de código, estos fragmentos son copiados literalmente en el fichero fuente (.hs). El primer fragmento se utiliza normalmente para declarar el nombre del módulo y las importaciones necesarias. Luego viene la especificación opcional de un *wrapper*, mas adelante comentaremos que es y para que sirve.

```
wrapper := '%wrapper' @string
```

Luego vienen las definiciones de macros, que agrupan ciertas expresiones regulares o conjuntos de caracteres en un único identificador. Por ejemplo:

```
$digit = 0-9
$alpha = [a-zA-Z]
```

Después llega el cuerpo principal donde se da forma a las posibles unidades léxicas reconocidas y se escriben entre corchetes las acciones adecuadas que se deban tomar en forma de reglas. Si no se desea hacer nada especial simplemente se termina con ';'.

```
$digit+ { \s ->Int (read s) }
$white+ ;
```

Para concluir, se escribe una parte de código Haskell donde va la función principal, así como cualquier tipo de función auxiliar que se desee utilizar en las reglas.

Resumiendo, Alex provee una interfaz básica para el analizador léxico generado, que puede utilizarse para reconocer unidades léxicas, dado un tipo abstracto de datos y unas operaciones sobre él mismo. También trae la posibilidad de incluir un *wrapper*, el cuál provee un nivel más alto de abstracción sobre la interfaz básica, esto es añadir en un nuevo tipo de datos superior más información acerca del texto analizado, que "envuelve" —de ahí el término *wrapper*— a la lista de *tokens* que se obtienen con el básico, el cuál no añade ninguna información. Alex posee varios *wrappers*, de los cuales se ha utilizado el llamado *posn* que añade algo más de funcionalidad que el básico, añadiendo información sobre la línea y la columna en la que se reconoce la unidad léxica en un tipo de datos llamado *AlexPosN*.

```
data AlexPosn = AlexPn !Int  -- desplazamiento absoluto del caracter
                       !Int  -- n'umero de l'inea
                       !Int  -- n'umero de columna
```

Esto es solo una aproximación a las posibilidades de Alex. Para más información se remite al lector a la dirección `http://www.haskell.org/alex/` donde encontrará toda la documentación necesaria.

3.2.3. Happy

Happy es un generador de analizadores sintácticos para Haskell, similar a la herramienta `Yacc` para C. Como `Yacc`, toma un archivo que contiene anotada una especificación BNF de una gramática y produce un módulo en Haskell que contiene el analizador sintáctico para la gramática.

Los usuarios de Yacc encontrarán Happy bastante familiar, la idea básica es la siguiente, escribir la gramática a analizar en un archivo con extensión `.y`, ejecutar Happy sobre él y utilizar la salida `.hs` como analizador integrado en el resto del programa. El analizador recibe normalmente como entrada una lista de unidades léxicas, en nuestro caso generadas por Alex.

Como ejemplo de uso y funcionamiento mostraremos la descripción de un analizador sintáctico de expresiones simples con enteros, variables, operadores y la forma **let** $var = exp$ **in** exp;. El fichero .y sería el siguiente:

Al principio del archivo hay una cabecera opcional del módulo, similar a la que comentamos antes para Alex:

```
{
module Main where
}
```

Después algunas declaraciones obligatorias:

```
%name calc
%tokentype { Token }
```

La primera línea declara el nombre de la función de análisis que Happy generará y la segunda declara el tipo de las unidades léxicas que el analizador aceptará.

A continuación se declaran todas los posibles unidades léxicas. Por ejemplo (no corresponden a nuestro compilador):

```
%token
  let           { TokenLet }
  in            { TokenIn }
  int           { TokenInt $$ }
  var           { TokenVar $$ }
  '='           { TokenEq }
  '+'           { TokenPlus }
  '-'           { TokenMinus }
  '*'           { TokenTimes }
  '/'           { TokenDiv }
  '('           { TokenOB }
  ')'           { TokenCB }
```

Los símbolos a la izquierda son las unidades léxicas como serán referidos en el resto de la gramática, y a la derecha y entre llaves el patrón Haskell que describe la unidad léxica de la que proviene. El símbolo `$$` es una función de proyección $f : \mathit{TokenInt}\ a \rightarrow a$ que obtiene el valor almacenado en la unidad léxica.

Como en `Yacc`, se incluye %% por ninguna razón en particular:
`%%`

Aquí tenemos las reglas de las producciones de las gramáticas:

```
Exp     : let var '=' Exp in Exp  { Let $2 $4 $6 }
        | Exp1                    { Exp1 $1 }

Exp1    : Exp1 '+' Term           { Plus $1 $3 }
        | Exp1 '-' Term           { Minus $1 $3 }
        | Term                    { Term $1 }

Term    : Term '*' Factor         { Times $1 $3 }
        | Term '/' Factor         { Div $1 $3 }
        | Factor                  { Factor $1 }

Factor  : int                     { Int $1 }
        | var                     { Var $1 }
        | '(' Exp ')'             { Brack $2 }
```

Cada producción consiste en un símbolo no terminal en la izquierda seguido de :, luego una o más alternativas para dicho símbolo a la derecha, separadas por | . Cada alternativa consta de la forma que puede tomar la parte derecha de la regla gramatical y la acción, en forma de código Haskell a realizar, entre {}.

El analizador reduce la entrada usando las reglas de la gramática en sentido ascendente hasta que sólo quede un símbolo, la raíz de la gramática, llamado `Exp` en el ejemplo. El analizador generado pertenece a la clase $LALR(1)$. El valor de este símbolo es lo que devuelve la función analizadora.

Para completar el programa necesitamos algo más de código, similar al código opcional permitido por Alex, con la función principal, la función de error y cualesquiera funciones auxiliares que sean necesarias:

{

```
happyError :: [Token] -> a
happyError _ = error ``Parse error''
```

También puede ir aquí el tipo de datos que se devuelve, o puede ser importado como en nuestro caso.

```
data Exp     = Let String Exp Exp
             | Exp1 Exp1

data Exp1    = Plus Exp1 Term
             | Minus Exp1 Term
             | Term Term

data Term    = Times Term Factor
             | Div Term Factor
             | Factor Factor

data Factor  = Int Int
             | Var String
             | Brack Exp
```

Y la estructura del tipo `Token`, en nuestro caso importada del módulo Alex. En el ejemplo, serían:

```
data Token
      = TokenLet
      | TokenIn
      | TokenInt Int
      | TokenVar String
      | TokenEq
      | TokenPlus
      | TokenMinus
      | TokenTimes
      | TokenDiv
      | TokenOB
      | TokenCB
      deriving Show
```

Finalmente la función principal, y las auxiliares necesarias:

```
    main = do s <- getContents
              print ((calc.lexer) s)
```

}

Estas son las reglas básicas del funcionamiento de Happy, para el lector interesado referimos el lugar donde está toda la documentación actualizada. `http://www.haskell.org/happy/`

3.2.4. *Pretty Printer*

Un *pretty printer* (o impresor amigable) es una herramienta, habitualmente implementada en una librería, que sirve para facilitar la impresión de árboles abstractos en un fichero de texto. En ella se ha definido un tipo de datos `Doc` que representa a un documento, junto con una serie de funciones llamadas **combinadores**, las cuales operan con uno o varios documentos para formar uno único con todos ellos.

Puesto que la librería es muy extensa, en esta sección se explicarán los combinadores que se han considerado para el desarrollo del módulo `Printer.hs`, descrito en la sección 5.2.

```
empty :: Doc
text  :: String -> Doc
```

La función `empty` representa a un documento vacío, y la función `text` convierte una lista de caracteres en un documento que las contenga. Observar que el documento vacío equivale a `text` .

```
(<>)  :: Doc -> Doc -> Doc
(<+>) :: Doc -> Doc -> Doc
(<$>) :: Doc -> Doc -> Doc
```

Los operadores <>, <+> y <$> son infijos, y sirven para concatenar documentos. El operador <> intercala entre ellos el documento vacío, el <+> un espacio en blanco y el <$> inserta un salto de línea entre ellos.

```
fillSep :: [Doc] -> Doc
vsep    :: [Doc] -> Doc
```

Los combinadores anteriores manejan listas de documentos. La primera de ellas inserta un espacio entre cada documento, equivalente a concatenarlos con el operador <+>. La segunda es similar solo que en este caso equivale a concatenarlos con <$>.

```
equals :: Doc
```

La función anterior representa al documento que contiene el símbolo '='. Aunque no tiene mucha relevancia, se utilizará en la implementación del módulo `Printer.hs`, por ello se explica su significado.

```
list   :: [Doc] -> Doc
tupled :: [Doc] -> Doc
```

Los combinadores anteriores sirven para, dada una lista de documentos, obtener una representación en forma de lista o tupla, respectivamente. Por ejemplo, dada la siguiente lista de documentos:

```
[text "doc1",text "doc2", text "doc3"]
```

El texto resultante de llamar a estas funciones sería el siguiente:

```
[doc1, doc2, doc3]
(doc1, doc2, doc3)
```

```
group :: Doc -> Doc
```

El combinador anterior deshace todos los saltos de línea contenidos de un documento dado. Si la línea resultante es menor que el ancho de página se escribe como tal. En caso contrario, el documento no se modifica.

```
parens :: Doc -> Doc
braces :: Doc -> Doc
```

Los combinadores anteriores sirven para encerrar documentos entre paréntesis '`(`' '`)`' y llaves '{' '}' respectivamente.

```
align :: Doc -> Doc
hang :: Int -> Doc -> Doc
```

El combinador `align` sirve para definir una columna de referencia que vendrá dada por la última palabra escrita antes del primer salto de línea. El combinador `hang` sangra un documento dado tantos espacios como indique el primer parámetro. Para ilustrar el uso de éstos con ejemplos, se remite al lector a la sección 5.2.

Capítulo 4

Fase de análisis léxico

En este capítulo describiremos el funcionamiento del analizador léxico. La misión de este módulo consiste en generar una lista de unidades léxicas a partir del fichero con el código fuente. Primero comentaremos las unidades léxicas, sus expresiones regulares y sus atributos. Posteriormente se describirá la implementación del analizador, las herramientas empleadas y sus reglas. Por último, se explicará el proceso de filtrado al que se somete la lista de unidades léxicas obtenida en primera instancia.

4.1. Unidades léxicas

A continuación mostraremos una tabla de las unidades léxicas disponibles en SAFE junto con una breve descripción de cada una de ellas. Para explicar las expresiones regulares asociadas a cada categoría léxica usaremos los siguientes conjuntos:

`dígito` → `[0−9]`
`minúscula` → `[a−z]` | `_`
`mayúscula` → `[A−Z]`
$\texttt{reservadas}_1$ → `let` | `where` | `case` | `data` | `of` | `in` | `_` | `self`
$\texttt{reservadas}_2$ → `|` | `!` | `− >` | `=` | `@`
$\texttt{reservadas}_3$ → `::`
$\texttt{reservadas}_4$ → `;` | `(` | `)` | `,` | `[` | `]` | `{` | `}` | `[]`
$\texttt{reservadas}_5$ → `True` | `False`
`reservadas` → $\texttt{reservadas}_1$ | $\texttt{reservadas}_2$ | $\texttt{reservadas}_3$ | $\texttt{reservadas}_4$ | $\texttt{reservadas}_5$
`símbolo` → `!` | `#` | `$` |`%` | `&` | `*` | `+` | `.` | `/` | < | `=` | > | `?` | `@` | \ | ∧ | `|` | −

En las expresiones regulares mostradas en el cuadro 4.1, los corchetes representan un conjunto, el caracter | la disyunción, el asterisco * la repetición, el signo + la repetición no nula, y los paréntesis angulares <> delimitan el conjunto que debe excluirse de las cadenas posibles.

Todas las unidades léxicas tienen un atributo en el que está almacenado la línea y la columna en la que se encontraban en el fichero fuente. Es necesario llevar estos dos parámetros porque, una vez analizado el texto fuente y obtenida la lista de los elementos léxicos, se realizará un proceso de filtrado que se explicará con detenimiento en la sección 4.3.

Por último, como es lógico, las unidades léxicas `Op`, `Id`, `CInfija`, `Constr`, `ConstNum` y `ConstBool` (mostradas en el cuadro 4.1) tienen un atributo adicional en el que se guarda el lexema o cadena de caracteres asociada.

Cuadro 4.1: El tipo Token

Unidad Léxica	Expresión Regular y descripción
Id	minúscula { minúscula | mayúscula | dígito | ' }* <reservadas$_1$> *Representa un identificador*
Op	símbolo { : | símbolo }* <reservadas$_2$> *Operador algebraico*
CInfija	: { símbolo | : }* <reservadas$_3$> *Constructora infija*
Constr	mayúscula { minúscula | mayúscula | dígito | ' }* <reservadas$_5$> *Constructora*
ConstNum	dígito$^+$ *Secuencia numérica*
ConstBool	reservadas$_5$ *Constantes booleanas*
Let	let *Expresión let*
Where	where *Definición local*
Case	case *Instrucción case*
Data	data *Declaración de estructuras de datos*
Of	of *Instrucción case*
In	in *Expresión let*
Guarda	| *Selector en las definiciones de función*
Excl	! *Determina si una estructura de datos es destruible o no*
Flecha	−> *Tipo funcional*
Igual	= *Definición*
At	@ *Permite copiar estructuras de datos en un espacio de memoria*
Def	:: *Signatura de tipos*
Semicolon	; *Separador de instrucciones*
Coma	, *Separador de elementos de una tupla*
ParIzq	(*Apertura de paréntesis. Asociatividad de una expresión*
ParDer	) *Cierre de paréntesis. Asociatividad de una expresión*
CorIzq	[*Apertura de corchete. Define listas*
CorDer	] *Cierre de corchete. Define listas*
BraIzq	{ *Apertura de bloque*
BraDer	} *Cierre de bloque*
ListaVacia	[ ] *Constructor de listas*
Subrayado	_ *Variable anónima*
Self	self *Región local*

4.2. Analizador léxico

Como se dijo antes, para la implementación del analizador léxico se ha utilizado la herramienta Alex. Con el propósito de mejorar el autómata que genera, en vez de describir una expresión regular por cada unidad léxica, es preferible no tratar las excepciones de las reglas como expresiones regulares aparte. Si se optara por escribir una regla por cada unidad léxica, Alex -por su propio funcionamiento que explicaremos más adelante- generaría un autómata con un número de estados demasiado elevado, y por tanto el rendimiento decrecería.

Veamos en un ejemplo cómo genera Alex un autómata y cómo se dispara su número de estados. Partamos de las reglas que hemos descrito para los identificadores y las palabras reservadas$_1$:

`id` → `minúscula` { `minúscula` | `mayúscula` | `dígito` | ’ }* <`reservadas`$_1$>
`reservadas`$_1$ → `let` | `where` | `case` | `data` | `of` | `in` | _ | `self`
`Ejemplos aceptados:` x, x', $tab1$, y, $\_\_$ `Ejemplos erróneos:` let, $\_$, $1x$, $'$

Si queremos reconocer como unidades léxicas diferentes los identificadores y los elementos de `reservadas`$_1$ cada uno como una unidad diferente, podríamos escribir una regla para cada una de ellas, para cada unidad léxica. En ese caso, Alex tendría que diferenciar multitud de situaciones, que en la práctica se traduce en generar un estado por cada nuevo carácter reconocido e ir comprobando si se ha alcanzado ya alguna palabra reservada o si, en cambio, todavía se está en un identificador. Si representamos cada estado por los caracteres reconocidos hasta el momento tendríamos:

‘‘l’’, ‘‘le’’, ‘‘let’’, ‘‘w’’, ‘‘wh’’, ‘‘whe’’, ‘‘wher’’, ‘‘where’’, ‘‘c’’, ‘‘ca’’,
‘‘cas’’, ‘‘case’’ ... ‘‘_’’, ‘‘s’’, ‘‘se’’, ‘‘sel’’, ‘‘self’’

Sólo los que coinciden con los elementos de `reservadas`$_1$ no reconocerían un identificador. Para solucionar este inconveniente, es preferible reconocer tanto los elementos de `reservadas`$_1$ como los identificadores en una misma regla (i.e. aplicar un filtro a la lista de unidades reconocidas). Si al realizar el análisis elemento a elemento nos encontramos con un identificador, debemos comprobar si corresponde con algún elemento de `reservadas`$_1$ y en ese caso modificarlo convenientemente para ajustarlo a su categoría léxica correspondiente.

Esta técnica es la que hemos adoptado en nuestro analizador. En concreto reconocemos como categorías léxicas iguales a:

- Los identificadores y las `reservadas`$_1$.
- Los operadores y las `reservadas`$_2$.
- Las constructoras infijas y la unidad léxica Def (‘‘::’’).
- Las constructoras y las `reservadas`$_5$.
- Todas las `reservadas`$_4$ juntas.

Esto quiere decir que cada vez que se analice un identificador se tiene que comprobar que no pertenece a `reservadas`$_1$, y si es así modificarlo. Si se trata de un operador, entonces no debe estar incluido en `reservadas`$_2$, de lo contrario, habrá que modificarlo. El procedimiento es análogo para el resto de igualdades enunciadas.

La implementación en Alex de las reglas ha sido la siguiente:

Definiciones auxiliares

```
$digito          = 0−9
$mayuscula       = [A−Z]
$minuscula       = [a−z \_]
$simbolo         = [\! \# \$ \% \& \* \+ \. \ \< \= \> \? \@ \\ \^ \| \−]
$simboloSinAt    = $simbolo # \@
$simboloSinExcl  = $simbolo # \!
$sinLlaves       = [\x00− \xff] # [\{ \}]
```

Expresiones Regulares

`$white+`	Se descarta la secuencia de caracteres blancos
``‘‘−−’’.*``	Se descartan por ser comentarios de línea
``‘‘{−’’ $sinLlaves* ‘‘−}’’``	Se descartan por ser comentarios de varias líneas*.
`$digito+`	Se acepta una unidad léxica ConstNum
`$minuscula [$minuscula $mayuscula $digito \’]*`	Se acepta la unidad léxica Id
`$simbolo ([\: $simboloSinAt] [\: $simbolo]*)*`	Se acepta un Op
`$simboloSinExcl [\: $simbolo]*`	Se acepta un Op
`\:[$simbolo \:]*`	Se acepta una CInfija
`$mayuscula [ $minuscula $mayuscula $digito \’]*`	Se acepta una Constr
`[\; \( \) \, \ \] \{ \}]`	Se acepta una Reservada_4
`\[\]`	Se acepta una Reservada_4

*Observar que en SAFE los comentarios de varias líneas no pueden contener ni una llave de apertura ni una llave de cierre.

Es interesante destacar la necesidad de utilizar dos reglas para reconocer operadores. En el cuadro 4.1 se describe la expresión regular de los operadores como:

`op` → `símbolo { : | símbolo }`* $<\text{reservadas}_2>$

`Ejemplos aceptados: @@, ==, +, <=, --,*` `Ejemplos erróneos: !, !@, ::`

Sin embargo, según esta expresión regular, podría derivarse el operador `!@`. Este no debe considerarse válido, puesto que la exclamación `!` y la arroba `@` ya son unidades léxicas por sí mismas —véase cuadro 4.1—. El problema reside en la preferencia que Alex otorga a las cadenas más largas frente a las de menor longitud. Si por ejemplo, se intentara reconocer la cadena de caracteres “`mivariable3`” no se reconocería por separado “`mivariable`” como un identificador y el “`3`” como una constante numérica, sino que se consideraría todo como un identificador “`mivariable3`”. La misma situación es la que sucede cuando se intenta reconocer la cadena “`!@`”, no obstante, en este caso no es deseable que se reconozcan como un todo, ya que en SAFE pueden darse expresiones de la siguiente forma:

```
x!@r
```

El significado de la expresión es que la estructura de datos contenida en x se copia a la región r, y a continuación x puede ser destruida. No significa que el operador `!@` toma como argumentos x y r. Por tanto debe evitarse que se pueda derivar el operador `!@` de la expresión regular asociada a los operadores. La idea que debe explotarse para conseguir este objetivo es que, si el operador comienza por un símbolo cualquiera, entonces el siguiente símbolo no puede ser una arroba, o, si el operador comienza por un símbolo que no sea la exclamación entonces el siguiente símbolo puede ser cualquiera. El equivalente en expresión regular es:

`Op -> símbolo { { : | símboloSinAt} { : | símbolo}`*`}`*

`Op -> símboloSinExcl { : | símbolo}`*

Para conocer en detalle la implementación a partir de la cual Alex genera la máquina de estados que hará las funciones de analizador léxico, se remite al lector al Apéndice A, donde se muestra el código fuente completo.

4.3. Postproceso

La fase del analizador léxico no devuelve la lista que reconoce el autómata sin más, sino que realiza una serie de procesos de filtrado y transformación a las unidades presentes en la lista generada por el autómata en primera instancia. En concreto son tres los procesos que deben acometerse sobre la lista de unidades léxicas, explicados en las siguientes subsecciones.

4.3.1. Inserción de los delimitadores de ámbito

El inicio de un ámbito se indica con la unidad `BraIzq`("{"), el fin por la unidad `BraDer`("}"), y las definiciones que contiene se separan a través de la unidad `SemiColon`(";").

En Haskell, para delimitar el ámbito de las definiciones se usa la regla del formato. En el lenguaje SAFE se hace un proceso similar al que se hace en el primero. Dicha regla se sirve de las columnas en las que se encontraban las unidades léxicas en el fichero fuente para discernir en qué ámbito se está. Siempre que una de estas comience un nuevo ámbito se guardará su columna en una pila, que será la referencia con la que se contará para comparar con la de la unidad actual y de este modo discernir en qué situación se encuentra. Estos ámbitos vienen delimitados por { } y las definiciones por ;. Pueden darse tres casos:

a) La unidad actual tiene una columna superior —más a la derecha en el texto fuente— a la de referencia. En este caso no sólo no sólo no se cambia de ámbito sino que se mantiene en la misma definición que en la que se encontraba la unidad anterior.

b) La unidad actual tiene la misma columna que la de referencia. Se encuentra dentro del mismo ámbito pero comienza una nueva definición. Por tanto se debe insertar en la lista de unidades léxicas un separador de definiciones: la unidad `SemiColon` (";").

c) La unidad actual tiene una columna inferior a la de referencia. Se ha concluido el ámbito anterior y por tanto se debe insertar un finalizador de bloque: la unidad `BraDer` ("}").

La lista de unidades léxicas que devuelve en primera instancia el analizador léxico debe ser por tanto procesada en función de estas reglas, aunque también debemos tener en cuenta algunos detalles adicionales: En SAFE, hay unidades léxicas que —independientemente de la columna actual y la de referencia—, advierten el inicio y el fin de los ámbitos de definiciones. A continuación se detallan estas unidades:

d) Las unidades léxicas `Of`("**of**"), `Where`("**where**") y `Let`("**let**") indican la aparición de un nuevo ámbito. Por tanto deberemos insertar tras ellos una marca de nuevo bloque, `BraIzq` ("{").

e) La unidad léxica `In`("**in**") indica el fin del ámbito actual. Se deberá, entonces, insertar una marca de cierre de bloque, `BraDer` ("}").

f) En SAFE, el programador puede escribir él mismo en el fichero fuente los delimitadores de ámbito o bloque (unidades `BraIzq` y `BraDer`) y los separadores de definición (unidad `SemiColon`). Consecuentemente, sólo habrá que insertarlos en la lista de unidades, cuando no estén ya en ella.

Cuadro 4.2: Categorías léxicas que se reconocen como iguales

Se reconocen como:	*Tokens*
Identificadores	Id, Let, Where, Case, Data, Of, In, Subrayado, Self
Operadores	Op, Guarda, Excl, Flecha, Igual, At
Constructoras	Constr, ConstBool
Constructoras infijas	CInfija, Def

Ejemplo:

```
funcion1 x = exp1
funcion2 y = exp2
funcion3 z = exp3 *
  exp4 *
        exp5
    exp6
funcion4 t =
  case t of
    alt1 -> p1
    alt2 -> p2
  where p1 = pp1
        p2 = pp2
```

```
{
funcion1 x = exp1;
funcion2 y = exp2;
funcion3 z = exp3 *
     exp4 *
            exp5
        exp6;
funcion4 t =
  case t of
    {
    alt1 -> p1;
    alt2 -> p2
    }
  where {
        p1 = pp1;
        p2 = pp2
        }
}
```

4.3.2. Distinción de las categorías léxicas reconocidas como iguales

Hay que recordar que la lista que se obtenía del analizador léxico en primera instancia no contenía toda la diversidad de unidades léxicas que debería, puesto que se optó por reconocer bajo el mismo patrón a categorías léxicas diferentes para reducir el número de estados del autómata. La implicación práctica de esta medida consiste en que según se va recorriendo la lista de unidades y realizando el filtrado en función de las reglas anteriores, se debe ir transformando cada elemento original en su unidad final. Es decir, si la unidad actual ha sido reconocida como una `Id` "**let**" es necesario aplicar la transformación necesaria para obtener una unidad `Let` y ejecutar las reglas pertinentes en función de la transformada y no de la original.

Ejemplo:

Si la lista que genera el autómata es la siguiente:
`[Id("let"), Id("x"), Constr("True"), CInfija("::"), CInfija(":")]`

Tras la transformación, que consiste en recalificar las palabras y símbolos reservados, se obtendría:
`[Let, Id("x"), ConstBool(True), Def("::"), CInfija(":")]`

Puesto que **let** y **True** son palabras reservadas y "`::`" es un símbolo reservado.

4.3.3. Extracción de la expresión principal

Por último, debido a la estructura de los programas en SAFE, debe aplicarse un último proceso a la lista ya completada con los delimitadores de ámbito y separadores de definiciones.
En SAFE, el programa debe tener la siguiente estructura:

```
Definiciones*
Expresión principal
```

Las definiciones pertenecen al mismo bloque mientras que la expresión principal va aparte en la estructura del programa. Sin embargo, puede suceder —siempre que el programador no delimite explícitamente las definiciones o que no escriba la expresión más a la izquierda que éstas— que la expresión principal esté a la misma altura que las definiciones (i.e. en la misma columna).

En esta situación, la única forma de saber cuál es la expresión principal, es recorrer la lista de unidades desde el final al principio para encontrar el último elemento que esté en la misma columna de referencia que la marcada por la primera definición.

Ejemplo:

Programa Original	Programa tras la inserción de los delimitadores de ámbito	Programa tras la extracción de la expresión principal
	{	{
def1	def1;	def1;
def2	def2;	def2;
...	...	...
defN	defN;	defN
expPPal	expPpal	}
	}	expPpal

Capítulo 5

Fase de análisis sintáctico

5.1. El árbol abstracto

Lo que que produce como resultado el analizador sintáctico será un árbol abstracto. Este mismo será utilizado como tipo de datos de comunicación entre las posteriores fases del compilador, así obtenemos un diseño modular e independiente, siendo este tipo de datos el único interfaz que tiene que ser respetado como entrada/salida de cada módulo.

A continuación se explica en detalle cada constructora del tipo de datos. Ver figura 5.1.

Programa Principal

Declaración de un programa. Está formado por una lista de declaraciones de tipos de datos, una lista de declaraciones de funciones y la expresión principal a evaluar.

```
type Prog a = ([DecData], [Def a], Exp a)
```

La variable de tipo 'a' contendrá las 'decoraciones' del árbol abstracto, que se usarán para almacenar cualquier tipo de información (compartición, tipos, ...).

Declaraciones de tipos de datos

Una declaración de un tipo de datos, contiene el nombre del tipo y una lista de variables y regiones y una lista de alternativas.

```
type DecData = (String, [VarTipo], [VarReg], [AltDato])

data AltDato = ConstrA String [ExpTipo] VarReg Constructora

data ExpTipo  = VarT VarTipo                              a
              | ConstrT String [ExpTipo] Bool [VarReg]    (Foo (a@r1))@r2. El booleano indica
                                                          si el tipo es destructivo o no.
              | Flecha ExpTipo ExpTipo                    t1 → t2
              | Rec                                       Aparición recursiva del tipo que se
                                                          está definiendo


type VarTipo = String
type VarReg = String
```

```
type Prog a   = ([DecData], [Def a], Exp a)
type DecData  = (String, [VarTipo], [VarReg], [AltDato])
data AltDato  = ConstrA String [ExpTipo] VarReg
data ExpTipo  = VarT VarTipo
              | ConstrT String [ExpTipo] Bool [VarReg]
              | Flecha ExpTipo ExpTipo
              | Rec
type VarTipo  = String
type VarReg   = String
type Def a    = ([ExpTipo], Izq a, Der a)
type Izq a    = (String, [(Patron a, Bool)], [VarReg])
data Der a    = Simple (Exp a) [Def a]
              | Guardado [(Exp a, Exp a)] [Def a]
data Exp a    = ConstE Lit a
              | ConstrE String [Exp a] VarReg a
              | VarE String a
              | ReuseE String a
              | App1E String [Exp a] a
              | App2E String [Exp a] VarReg a
              | LetE [Def a] (Exp a) a
              | CaseE (Exp a) [(Patron a, Exp a)] a
              | CaseDE (Exp a) [(Patron a, Exp a)] a
data Lit a    = LitN Int
              | LitB Bool
```

Figura 5.1: Sinopsis del Árbol Abstracto

Declaraciones de funciones

Definición de una función:

```
type Def a = ([ExpTipo], Izq a, Der a)
```

Comprende las sentencias de la forma:

Izq = Der

También incluye el tipo de la definición proporcionado por el usuario (si lo hay).

```
type Izq a = (String, [(Patron a, Bool)], [VarReg])
```

Parte izquierda de una declaración: nombre pat_1 ... pat_n @ reg_1 ... reg_n
El booleano asociado a cada patrón indica si contiene el símbolo ! o no.
La lista de regiones indica la región donde se guardará la DS creada por la función, siendo en la ultima región donde se almacenará la estructura más externa. En el caso de que no se genere ninguna, la lista será vacía.

Patrones

Los patrones en las partes izquierdas de las definiciones muestran la forma en la que deben encajar los argumentos para que se ejecute la expresión de la parte derecha, estos patrones pueden ser constantes literales, variables o constructoras de datos. Los patrones lista $([], p1 : p2)$ y los patrones tupla $(p_1, \ldots, p_n)$ se consideran un caso especial de patrón con constructora (ConstrP).

```
data Patron a =
ConstP Lit                      Patrón literal (Entero o Bool)
| VarP String a                 Patrón variable (x)
| ConstrP String [Patron a] a   Patrón tipo construido (C p1...pn)
```

Parte derecha de una declaración:

```
data Der a
```

Hay dos alternativas:

Declaración simple: sólo contiene una expresión y opcionalmente, cláusulas **where**:

```
x + 1 where x = 3
```

Declaración con guardas y opcionalmente, cláusulas **where**:

```
 | x <= y = (x:(merge xs ys)@r)@r
 | x >  y = (y:(merge xs ys)@r)@r
 where x = ...
```

```
Simple (Exp a) [Def a]
Guardado [(Exp a, Exp a)] [Def a]
```

Expresiones

Las expresiones se componen de diferentes formas, desde simples constantes literales hasta cláusulas let y case pasando por aplicaciones de funciones y elementos mas propios de SAFE como las copias y el reuso de variables o el case destructivo.El tipo 'a' que incluyen se podrá almacenar la información de compartición, información de tipo, etc..

`data Exp a = ConstE Lit a`	Literal (Entero o Bool)
`| ConstrE String [Exp a] VarReg a`	Construcción de DS: $(Ce_1...e_n)@r$
`| VarE String a`	Variable: x
`| CopyE String VarReg a`	Copia de DS: $x@r$
`| ReuseE String a`	Reutilización de DS: $x!$
`| App1E String [Exp a] a`	Llamada a función que no construye DS: $fe_1...e_n$
`| App2E String [Exp a] VarReg a`	Llamada a función que construye DS: $(fe_1...e_n)@r$
`| LetE [Def a] (Exp a) a`	Let: let def_i in e
`| CaseE (Exp a) [(Patron a, Exp a)] a`	Case: case e of $p_i \rightarrow e_i$
`| CaseDE (Exp a) [(Patron a, Exp a)] a`	Case destructivo: case! e of $p_i \rightarrow e_i$

Literal: entero o booleano

```
data Lit =
 LitN Int
 LitB Bool
```

5.2. Impresión amigable del árbol

Para visualizar un programa representado mediante el árbol abstracto especificado en la sección anterior, es necesario desarrollar un módulo que lo procese y devuelva como resultado un fichero de texto con una representación del mismo "amigable" para el usuario. Para ello se ha utilizado la

librería *pretty printer* explicada en la sección 3.2.4.

La idea es la siguiente: definir una función por cada declaración **data** del árbol, mediante una expresión **case** con tantas alternativas como las del dato correspondiente. Si alguna de estas contiene a su vez otras alternativas, se llamará a la función que le corresponda. En caso contrario, se indicará de qué forma se quiere mostrar la información de esa alternativa, utilizando los combinadores que se consideren adecuados para facilitar la visualización de la misma. En algunas funciones habrán alternativas adicionales para distinguir casos específicos, que detallaremos posteriormente con algunos ejemplos.

Para ilustrar esta idea, se muestra a continuación algunos fragmentos del código que implementa la impresión del árbol abstracto:

```
imprimeProg prog = case prog of
  ([], df, exp) -> imprimeDefs df <$> empty <$> imprimeExp exp
  (dd, df, exp) -> imprimeDecsDato dd <$> empty <$> imprimeDefs df <$> empty <$> imprimeExp exp
```

Como se ha visto en la sección anterior, un programa `Prog` se define mediante sólo una alternativa. Sin embargo, puesto que el programador no está obligado a incluir declaraciones de tipo, se han definido dos alternativas. La primera de ellas sirve para que no se impriman líneas en blanco innecesarias, mientras que la segunda es la que se espera encontrar en la definición de `imprimeProg`, según el esquema explicado anteriormente.

```
imprimeExpTipo e = case e of
  VarT vt -> text vt
  ConstrT "TuplaT" ets False vrs -> group (tupled (map imprimeExpTipo ets)) <+> text "@"
                                    <+> imprimeIds vrs
  ConstrT "TuplaT" ets True vrs -> group (tupled (map imprimeExpTipo ets)) <> text "! @"
                                   <+> imprimeIds vrs
  ConstrT "ListaT" ets False vrs -> group (list (map imprimeExpTipo ets)) <+> text "@"
                                    <+> imprimeIds vrs
  ConstrT "ListaT" ets True vrs -> group (list (map imprimeExpTipo ets)) <> text "! @"
                                   <+> imprimeIds vrs
  ConstrT s [] _ [] -> text s
  ConstrT s ets True vrs -> parens (text s <+> imprimeExpTipos ets <+> text "! @"
                                    <+> imprimeIds vrs)
  ConstrT s ets False vrs -> parens (text s <+> imprimeExpTipos ets <+> text "@"
                                     <+> imprimeIds vrs)
  Flecha et1 et2 -> imprimeExpTipo et1 <+> text "->" <+> imprimeExpTipo et2
  Rec -> text "REC"
```

La función anterior se corresponde con la impresión de una expresión de tipo `ExpTipo`. Aunque este último sólo tiene cuatro alternativas, se han distinguido los casos en los que se construyen listas y tuplas, así como aquellos en los que se indica que un parámetro es destruible o no, además de los casos en los que una lista dada pueda ser vacía o no. Para las listas y las tuplas se han utilizado los combinadores `list` y `tupled` respectivamente.

```
imprimeExp exp = case exp of
  ConstE lit a -> imprimeLit lit <+> braces (text "::" <+> (imprimeExpTipo a))
  ConstrE s [] vr a -> text s <+> text "@" <+> text vr
                       <+> braces (text "::" <+> (imprimeExpTipo a))
  ConstrE s exps vr a -> text s <+> imprimeExpsP exps <+> text "@" <+> text vr
                         <+> braces (text "::" <+> (imprimeExpTipo a))
  VarE s a -> text s <+> braces (text "::" <+> (imprimeExpTipo a))
  CopyE s vr a -> text s <+> text "@" <+> text vr <+> braces (text "::" <+> (imprimeExpTipo a))
  ReuseE s a -> text s <> text "!" <+> braces (text "::" <+> (imprimeExpTipo a))
  App1E s exps a -> text s <+> imprimeExpsP exps <+> braces (text "::" <+> (imprimeExpTipo a))
```

```
App2E s exps vr a -> text s <+> imprimeExpsP exps <+> text "@" <+> text vr
                     <+> braces (text "::" <+> (imprimeExpTipo a))
LetE ds exp a -> align (text "let" <+> imprimeDefsLet ds <$> text "in" <+> imprimeExp exp)
                        <+> braces (text "::" <+> (imprimeExpTipo a))
CaseE e alts a -> hang 2 (text "case" <+> imprimeExp e <+> text "of" <$> imprimeAltsCase alts)
                          <+> braces (text "::" <+> (imprimeExpTipo a))
CaseDE e alts a -> hang 2 (text "case!"<+> imprimeExp e <+> text "of" <$> imprimeAltsCase alts)
                           <+> braces (text "::" <+> (imprimeExpTipo a))
```

El fragmento anterior se corresponde a la función que imprime expresiones. En este caso se comprueba que se ha definido una alternativa para cada una de las posibles definidas en `Exp` (a excepción de la `ConstrE`, donde se distingue el caso particular de la lista vacía en el segundo parámetro). Cabe destacar que en esta función se ha decidido imprimir las decoraciones del árbol entre llaves para facilitar la comprensión de la impresión resultante. En cada alternativa se puede observar como se trata de asemejar la impresión a un formato adecuado (si no, el ideal) en el que se deberían escribir los programas en SAFE.

Por ejemplo, en las expresiones **let** se ha decidido alinear la palabra "**let**" con "**in**" usando el combinador `align`, imprimiendo además cada definición en una línea aparte, de modo que la columna de referencia se distinga con claridad. El resultado sería el siguiente:

let def_1
  ...
  def_n
in *exp*

En las expresiones **case** se ha decidido sangrar las alternativas usando el combinador `hang`. Cada alternativa se imprime en una línea aparte, quedando como sigue:

case *exp* **of**
  $alt_1 \rightarrow exp_1$
  ...
  $alt_n \rightarrow exp_n$

De este modo, se comprueba que siguiendo la jerarquía del árbol que se desea imprimir, y mediante la idea que se ha explicado anteriormente, se puede obtener una impresión amigable mediante una implementación sencilla y fácilmente ampliable.

En el ejemplo del capítulo 8 se ha utilizado este módulo para visualizar los resultados de cada fase, donde se comprueba que la salida se asemeja en la medida de lo posible al programa original.

5.3. Analizador sintáctico

Para implementar el analizador sintáctico hemos utilizado la herramienta `Happy`, el equivalente a `Yacc` o `Bison` en su versión para Haskell. En la sección 3.2.3 se describe su funcionamiento. Aquí solo comentaremos su código fuente, el fichero .y.

En la primera sección se describen las unidades léxicas (en adelante UL) que vamos a utilizar como símbolos terminales y sus equivalentes de la lista de ULs que nos ha proporcionado el análisis léxico. En la segunda sección, tras el separador "`%%`" se pasan a especificar las categorías sintácticas así como las acciones que tiene que ir llevando a cabo. En nuestro caso se contemplan unas 40 categorías diferentes que generan un autómata de unos 200 estados y utiliza unos 50 símbolos terminales. Por lo amplio del fichero y dada la cantidad de categorías sintácticas auxiliares sólo vamos a comentar las más importantes. Una nota a destacar es que utilizamos el constructor unit "`()`" para rellenar el espacio de *decoración* de los diferentes elementos del árbol abstracto que lo

necesitan ya que en este paso no se hace uso de ello. También explicamos el uso de la notación que hemos utilizado para las listas de elementos, que es el nombre de la categoría en plural seguido, en el caso que sea necesario, del separador que describe la forma (i.e. `pat, pats, patsComa`). Por otro lado, uno de los procesos que se han dado en algunas de las categorías sintácticas más importantes y complejas es la jerarquización de la gramática de forma que, aunque construyan el mismo tipo de datos, el analizador los trate de diferente forma evitando ambigüedades o conflictos desplazamiento-reducción. En este caso la notación no sigue un estándar entre las diferentes categorías, aunque sí siguen las mismas normas a grandes rasgos. La categoría sintáctica en el orden más superior recibe el nombre general mientras que según vamos bajando se les van asignando el mismo nombre seguido de letras o números para reflejar la condición de anidamiento, y en algunos casos se les da un nombre diferente para reflejar una formación muy específica (i.e. `aplic` es la categoría que describe las aplicaciones de funciones pero está dentro de la jerarquía de las expresiones). Otro caso de categorías sintácticas con una notación parecida a otras son las que llamamos categorías restringidas, que no son más que un subconjunto de la categoría sintáctica original que es válido para la situación en cuestión, generalmente referida en el nombre (i.e. `defsLet` es un subconjunto de `defs` que contiene las formas de definiciones que son apropiadas o válidas para las definiciones de una construcción `let`).

Sin más notas de consideración vamos a ver algunas de las partes más importantes del fichero.

```
prog  :: {Prog ()}
prog  : '{' defs '}' exp  {separaListas (reverse $2) $4}
      | exp
```

`Prog` es la categoría sintáctica principal desde la cuál comienza el análisis, vemos que esta formado por una lista de definiciones entre llaves opcional y una expresión principal, estas definiciones son de diferentes tipos que luego clasificaremos y ordenaremos en el postproceso, como se ve en el ejemplo utilizamos un `Maybe` de un `Maybe` para irlas separando según son siendo analizadas y en dicho postproceso posterior separarlas fácilmente.

```
def  :: {Definicion ()}
def  : defFun            {Right (Right $1)}
     | defData           {Left $1}
     | decTipo           {Right (Left $1)}
```

Estas definiciones son también una parte importante del compilador. Trataremos primero la que describe las definiciones de nuevos tipos de datos. Todas ellas construyen objetos del tipo de dato `Definicion a`.

```
defData  :: {DecData}
defData  : data constr '@' ids '=' altDatos          {((fst $2),[],$4,$6)}
         | data constr ids '@' ids '=' altDatos      {((fst $2),$3,$5,$7)}

altDato  :: {AltDato}
altDato  : constr '@' id                             {ConstrA (fst $1) [] (fst $3)}
         | constr expTiposc '@' id                   {ConstrA (fst $1) $2 (fst $4)}
         | '(' expTipoc cinfija expTipoc ')' '@' id  {ConstrA (fst $3) [$2,$4] (fst $7)}
```

Vemos como estas dos categorías analizan la forma adecuada la entrada y construyen los elementos del tipo de datos apropiado en cada momento, en `altDato` vemos la aparición de `expTipoc`, una de las formas jerárquicas de las expresiones de tipo, el por qué de esta jerarquización es referente a la ambiguedad natural de la gramática, así que las separamos haciendo que las más profundas sean más prioritarias logrando la precedencia y el asociamiento deseado. Por la longitud de estas defi-

niciones referimos al lector al apéndice B donde se muestra el código del fichero `.y`. Las categorías en cuestión son: `expTipo`, `expTipob`, `expTipoc` y sus correspondientes plurales, pero para su mejor entendimiento proponemos los siguientes ejemplos.

Int → T1 [a] @ ρ_2 b @ ρ_1 ⟹ (Int → (T1 ([a] @ ρ_2) b @ ρ_1))

f x (y * z) → (f (x (y*z)))

let x = x + y **in** x + 5 =¿(**let** (x = (x + y)) **in** (x + 5))

Otras definiciones importantes son las declaraciones de tipo de una función, no ya por su forma, que resulta ser muy simple,

```
decTipo  :: {DecTipo}
decTipo  : id ‘‘::’’ expTipo  {((fst $1),$3)}
         | op ‘‘::’’ expTipo  {((fst $1),$3)}
```

si no por la morfología del árbol abstracto en el que no hay ninguna constructora referida a esta definición si no que se fusiona, como se explica en el apartado 5.4, con la definición de la función correspondiente.

```
defFun  :: {Def ()}
defFun  : patFun der                    {([],(‘‘PAT’’,[($1,False)],[]),$2)}
        | id patsd der                  {([],((fst $1),$2,[]),$3)}
        | id der                        {([],((fst $1),[],[]),$2)}
        | id patsd ’@’id der            {([],((fst $1),$2,[(fst $4)]),$5)}
        | id ’@’ id der                 {([],((fst $1),[],[(fst $3)]),$4)}
        | patd op patd der              {([],((fst $2),[$1,$3],[]),$4)}

der     :: {Der ()}
der     : ’=’exp                        {Simple $2 []}
        | guards                        {Guardado $1 []}
        | ’=’ exp where ’{’ defsFun ’}’  {Simple $2 $5}
        | guards where ’{’ defsFun ’}’   {Guardado $1 $4}
```

Aquí vemos cómo la parte izquierda está formada por el identificador de la función, una serie de patrones, región si la función construye algo, y una parte derecha formada por expresiones con una serie de cláusulas **where** opcionales. Lo más importante de todo esto es el siguiente nivel sintáctico, el de las expresiones y los patrones, las cuales están de nuevo jerarquizadas y a la vez siguiendo un paralelismo natural debido al que se da entre la parte izquierda y derecha de una definición, también esta vez la jerarquía corresponde a cómo deseamos que se desambigue la gramática, de manera similar a las expresiones de tipo. De nuevo referimos al lector a los apéndices donde podrá encontrar estas categorías sintácticas en toda su extensión: `pat`, `pat1`, `pat2` y `pat3` para los patrones y `exp`, `exp1`, `exp2` y `aplic` para las expresiones, todas ellas con sus correspondientes plurales, para mayor claridad damos un ejemplo de asociatividad según la jerarquía adoptada.

foo (x:xs) (C1 a b @ r) ≠ foo (x:xs) (C1 (a b) @ r).

5.4. Postproceso

Todo el postproceso está centrado en solucionar un problema que ya se ha comentado anteriormente, el hecho de que en la categoría sintáctica del programa (Prog a) no se imponga un orden entre los tipos de declaraciones de datos y definiciones de función o de tipos definidos por el usuario ya que al no estar impuesto ningún orden entre ellas podamos tener un archivo fuente totalmente flexible en este sentido y podamos importar otros ficheros fuentes como módulos o librerías. En este caso se trata de separar y fusionar las diferentes listas para que, por un lado queden separadas las definiciones de nuevos tipos de datos y por otro se traten en conjunto las declaraciones de tipo y las definiciones de las funciones, para esto utilizamos la facilidad que nos da el tipo `Maybe` como hemos comentado antes.

Por otro lado, que en el árbol abstracto no haya una constructora para las declaraciones del tipo de las funciones si no que vaya insertada en la definición de la función supone otro problema. En este punto y como más adelante se verá en el analizador semántico, vamos a carecer de la capacidad de saber donde apareció, si es que lo hizo, la declaración del tipo de la función, aparte de fusionar la declaración de tipo con la primera aparición de la definición de la función introducimos un elemento semántico que se da cuenta de que éstas estaban formuladas en el orden correcto, es decir las diferentes definiciones seguidas de la declaración de tipo y todas ellas seguidas. Para lograr esto y luego no confundir una situación de no declaración de tipo con una declaración sin definiciones o viceversa en el caso de que no exista la declaración de tipo correspondiente, la primera definición se señala con una marca especial, imposible de introducir por el programador, y que el analizador semántico reconocerá apropiadamente en el siguiente paso del análisis. Para una comprensión mas profunda del funcionamiento exacto de estos procesos referimos al lector al apéndice B, más concretamente a la parte final del fichero `.y` donde están implementadas y exhaustivamente comentadas estas funciones.

Capítulo 6

Fase de comprobación semántica

En este capítulo se describe la fase de análisis semántico y de renombramiento de identificadores. Esta fase está definida mediante reglas de deducción, cuya notación aclararemos, para cada uno de los tipos principales de construcción del arbol abstracto del lenguaje, si bien al final se interrelacionan unas con otras.

En estas reglas también se renombran los identificadores dentro de su ámbito de uso como también veremos. Y por último daremos unas explicaciones sobre las partes mas importantes del fichero fuente encargado del trabajo.

6.1. Comprobaciones semánticas

Como ya se ha comentado en la introducción estas comprobaciones están basadas en reglas de deducción relacionadas entre sí. Para una mejor comprensión de las mismas se explicará la notación utilizada y se separarán por ámbitos. Si en algún caso no se llega a entender algún renombramiento de variable, todos se aclararán en el siguiente apartado. Todas estas reglas implementan las siguientes ideas informales:

- Identificación / renombramiento de identificadores:
    - No debe haber dos apariciones de definición con el mismo nombre en el mismo ámbito.
    - Toda aparición de definición y sus correspondientes apariciones de uso, será renombrada, siendo sustituida por un nombre fresco.
    - Toda aparición de uso ha de corresponder a alguna aparición de definición en ese ámbito o en otros más externos.
    - No hay ámbitos anidados de región. Tanto las apariciones de definición como las de uso están en categorías sintácticas diferenciadas de los identificadores. Por ello, se permiten nombres que solapen con los de otros identificadores.
    - Los nombres de funciones, tipos de datos y constructoras de valores son globales a todo el programa.
    - En las declaraciones **data** de tipos definidos por el usuario se exige además que toda aparición de definición tenga, al menos, un uso. Además, las apariciones recursivas deben tener los mismos argumentos que en la aparición de definición y la región externa debe ser la misma en todas las alternativas e igual a la más externa de la declaración de tipo del usuario, es decir, la última.

- Otras comprobaciones:

  - Todas las ecuaciones de definición de una función deben aparecer consecutivas.
  - La declaración explícita del tipo de una función, si existe, debe aparecer antes de la primera ecuación de definición.
  - Las definiciones que aparecen dentro de sentencias **let** y **where** deben ser restringidas a la siguiente forma: *patrón = expresión.*

En la figura 6.1 se describe la notación que se utilizará para definir las reglas de deducción, mostradas en las figuras 6.2, 6.3, 6.4 y 6.5.

6.2. Ámbito de los identificadores

En Haskell, y también en la sintaxis dulce de SAFE, no es necesario para el programador utilizar variables diferentes para cada ámbito pudiendo utilizar los mismos identificadores durante todo el programa, si bien a más bajo nivel sí se requiere una distinción de estos identificadores para los posteriores análisis. Por ello renombramos los identificadores sin perder la coherencia en el mismo ámbito, como se puede ver en las reglas anteriores. Otra pregunta que surge es cuándo se crea un ámbito. Esta es fácilmente resuelta fijándonos en las reglas que crean un nuevo ámbito en la tabla TS , con la operación $T : TS$. Veamos cada uno de estos casos:

Las expresiones **let** y **case** introducen un nuevo ámbito siendo el renombramiento de los patrones el que deben adoptar los identificadores correspondientes. Ejemplo:

```
f x y z = let x = 1 in x
```

su renombramiento es

```
f x y z = let x' = 1 in x'
```

En el renombramiento de patrones vemos como las cláusulas **where** generan un nuevo ámbito con los patrones de la primera ecuación, el cual anidan con los de la segunda, y así sucesivamente hasta que se renombre la expresión con todos los identificadores renombrados. La explicación para esto, como se verá más adelante en el Capítulo 7, es que un conjunto de definiciones **where** equivale a una secuencia de **let** anidados.

Por último, cada definición global genera un ámbito nuevo a nivel global y que a su vez se propaga en las partes izquierda, para la cual se crea un nombre fresco para cada aplicación de definición y cada una de las veriables ha de ser distinta de las demás, de ahí que la operación de unión de tablas de ámbito sea exclusiva.Y todo ello se propaga a la parte derecha de las ecuaciones. Un detalle importante es que para estas definiciones se pueden utilizar dos reglas de las mostradas en la figura 6.4. La primera es para la primera aparición, como se ve f todavía no pertenece a T_0 y devuelve f para luego ser insertada en T_0. En la segunda se comprueba que ya esté definida y se devuelve el conjunto vacío ya que no hay información nueva que suministrar a T_0.

Todo esto se ve de forma mucho más claro en los ejemplos, por lo que remitimos al lector al Capítulo 8 en el que se desarrolla el ejemplo completo.

6.3. Analizador semántico

El programa utilizado para realizar todas estas funciones es lo que hemos llamado Analizador Semántico y sus fuentes se muestran en el Apéndice C. Lo más destacado de este código y lo que

causa que no quede mucho que comentar sobre él, es su aproximación casi exacta a las reglas descritas anteriormente, ya que al estar implementadas en un lenguaje funcional como Haskell, son simples, aunque no siempre triviales, "traducciones" de las reglas al lenguaje funcional, poniendo como condiciones las precondiciones de las reglas y como valor devuelto la postcondición adecuada. Aparte claro está de la implementación de las operaciones requeridas, realizadas de forma muy natural al haberse especificado como listas, lo que se acercan mucho a su implementación real.

Como guía para el lector comentaremos las diferentes reglas mostradas en las figuras. En primer lugar las reglas para expresiones de la figura 6.2. No hay renombramiento de literales enteros y booleanos verificandose automáticamente, así como la validación de la región *self*. Las formas básicas, como son las variables la copia y el reuso simplemente buscan el renombramiento en la tabla y lo devuelven. El caso de **let** y **case** es algo más complicado, primero comprueban la expresión principal y luego los patrones para crear nuevos ámbitos que servirán de referencia para la aplicación de las reglas a las expresiones correspondientes. Para las expresiones guardadas simplemente se aplican las reglas a un lado y a otro y se devuelven las expresiones renombradas.

Reglas para los patrones, ver 6.3. Las constantes se validan automáticamente y para las variables se crea un nombre fresco que es apuntado en la tabla de salida, las constructoras son un paso recursivo aplocando a cada patrón de la constructora la trasnformación y devolveindo la unión de todas las tablas de salida. La regla para renombramiento de patrones en uan cláusula **where** ha sido explicada en la sección anterior.

En la figura 6.4 sólo queda comentar que para varias definiciones se vantransformando en orden utilizando como T_0 para la siguiente la T_0 original unida a la T_0' de salida de la definición actual.

La regla para los tipos de datos declarados por el usuario, ver 6.5 es una traducción literal de la regla informal dada anteriormente para este efecto.

Para más información, señalar que las relaciones de renombramiento y validación de declaraciones de datos, expresiones, patrones, definiciones globales, partes izquierdas y partes derechas se llaman respectivamente, `renombraData`, `renombraExp`, `renombraPatron`, `renombraDefs`, `renombraIzq` y `renombraDer`, siendo estas las funciones más importantes del programa, a la par que las más simples, mientras que el resto son funciones auxiliares que tratan con problemas concretos o con particularidades del lenguaje de implementación.

En caso de que un programa dado no cumpla las restricciones semánticas especificadas, el proceso de compilación se interrumpirá, informando al programador de que hubo un error semántico, identificando la restricción violada.

TS	Tabla de símbolos anidada. Puede verse como una lista $\mathrm{T}_m : \mathrm{T}_m\text{-}1 : ... : \mathrm{T}_0$, donde a mayor número más interno es el ámbito, siendo T_m el ámbito actual.
T_0	Tabla global de TS, donde se apuntaran las definiciones a nivel del programa.
TR	Tabla de regiones de la función en curso. Sirve para validar que la región sobre la que se construye algún dato está efectivamente declarada. La región local y temporal *self*, siempre validará positivamente cualquier consulta.
$T : TS$	Acción de añadir un nuevo ámbito T a TS.
$[x \mapsto x']$	Tabla unitaria donde x es la clave, variable del programa, y x' es su renombramiento, correspondiente a un nombre fresco.
$\bigcup_i T_i$	Operación de unión de todas las tablas de un ámbito. Esta unión es exclusiva, es decir, sólo está definida si no hay dos claves iguales.
$TS(x) = x'$	Operación de obtención de renombramiento. El identificador x está definido en algún ámbito, recorrido de interno a externo, de TS y su renombramiento es x'.
$UsoDer(d)$	Comprobación de uso. Dada una declaración de tipo del usuario, comprueba que todos los identificadores utilizados en la definición tienen, al menos, una aparición de uso.
$T_0' = Rec(T_0, d)$	Operación de transformaciones recursivas. Dada una declaración d comprueba si existe alguna aparición recursiva con los mismos parámetros, en ese caso devuelve una tabla T_0' que es T_0 cambiándole las apariciones recursivas por el símbolo especial `REC` utilizado posteriormente en el análisis de compartición
$T_0 \vdash C, T_0 \vdash f$	Equivale a $C \in dom(T_0), f \in dom(T_0)$.
$T_0 \vdash_{dd} d \mid T_0'$	Comprobación de declaraciones de datos del usuario. Los identificadores de la declaración d están bien definidos y además todos son utilizados al menos una vez en la tabla derecha, las apariciones recursivas de d se hacen sobre los mismos argumentos y las regiones externas de todas las alternativas son la misma. T_0' es T_0 añadiéndole la información que da la declaración.
$TS \vdash e \mid e'$	Renombramiento de expresiones. Los identificadores de la expresión e están bien definidos en TS y e' es el resultado de sustituir en e dichos símbolos por sus renombramientos.
$TS \vdash_d p \mid p', T$	Renombramiento de patrones. Los identificadores del patrón p están bien definidos con respecto a TS, el renombramiento es p' y se genera una tabla de un nuevo ámbito T.
$T_0 \vdash_g def \mid def', T, T_0'$	Renombramiento de definiciones globales. La definición global def está bien construida con respecto a la tabla global T_0, *def'* es su renombramiento y se generan dos tablas de ámbito: T de un ámbito más interno y T_0' añadiéndole a T_0 los nuevos identificadores globales.

Figura 6.1: Notación para las reglas de deducción

$$\frac{}{TS \vdash i \mid i} \qquad \frac{}{TS \vdash b \mid b} \qquad \frac{TS(x) = x'}{TS \vdash x \mid x'} \qquad \frac{TS(x) = x'}{TS \vdash x! \mid x'!}$$

$$\frac{}{T_R \vdash self} \qquad \frac{TS(x) = x' \ , \ T_R \vdash r}{TS \vdash x@r \mid x'@r}$$

$$\frac{T_0 \vdash f \quad \forall_i \ (\ TS \vdash e_i \mid e'_i \ ) \quad T_R \vdash r}{TS \vdash f \ \overline{e}_i@r \mid f \ \overline{e'}_i@r} \qquad \frac{T_0 \vdash C \quad \forall_i \ (\ TS \vdash e_i \mid e'_i \ ) \quad T_R \vdash r}{TS \vdash C \ \overline{e}_i@r \mid C \ \overline{e'}_i@r}$$

$$\frac{TS \vdash e_1 \mid e'_1 \quad TS \vdash_d p_1 \mid p'_1 \ , \ T_1 \quad T_1 : TS \vdash e \mid e'}{TS \vdash \textbf{let } p_1 = e_1 \textbf{ in } e \mid \textbf{let } p'_1 = e'_1 \textbf{ in } e'}$$

$$\frac{TS \vdash e \mid e' \quad \forall_i \ (\ TS \vdash_d p_i \mid p'_i \ , \ T_i \quad T_i : TS \vdash e_i \mid e'_i \ )}{TS \vdash \textbf{case } e \textbf{ of } \overline{p_i \rightarrow e_i} \mid \textbf{case } e' \textbf{ of } \overline{p'_i \rightarrow e'_i}}$$

$$\frac{\forall_i \ (\ TS \vdash e_i \mid e'_i \quad TS \vdash e''_i \mid e'''_i}{TS \vdash \overline{\mid e_i = e''_i} \mid \overline{\mid e'_i = e'''_i}}$$

Figura 6.2: Reglas semánticas para expresiones

$$\frac{}{TS \vdash_d i \mid i \ , \ \emptyset} \qquad \frac{}{TS \vdash_d b \mid b \ , \ \emptyset} \qquad \frac{fresca(x')}{TS \vdash_d x \mid x' \ , \ [x \mapsto x']}$$

$$\frac{\forall_i \ (\ TS \vdash_d p_i \mid p'_i \ , \ T_i \ ) \quad T_0 \vdash C}{TS \vdash_d C \ \overline{p_i} \mid C \ \overline{p'_i} \ , \ \cup_i T_i}$$

$$\frac{\begin{array}{c} TS \vdash e_1 \mid e'_1 \quad TS \vdash_d p_1 \mid p'_1 \ , \ T_1 \quad T_1 : TS \vdash e_2 \mid e'_2 \ \ldots \ T_{n-1} : \ldots : T_1 : TS \vdash e_n \mid e'_n \\ T_{n-1} : \ldots : T_1 : TS \vdash_d p_n \mid p'_n \ , \ T_n \quad T_n : \ldots : T_1 : TS \vdash e \mid e' \end{array}}{TS \vdash e \textbf{ where } \overline{p_i = e_i} \mid e' \textbf{ where } \overline{p'_i = e'_i}}$$

Figura 6.3: Reglas semánticas para patrones

$$\frac{T_0 \not\vdash f \quad \forall_i \ (\ T_0 \vdash_d p_i \mid p'_i \ , \ T_i)}{T_0 \vdash_g f \ \overline{p_i}@r \mid f \ \overline{p'_i}@r \ , \ (\cup_i T_i) \cup [r \mapsto ()] \ , \ [f \mapsto ()]}$$

$$\frac{T_0 \vdash f \quad \forall_i \ (\ T_0 \vdash_d p_i \mid p'_i \ , \ T_i)}{T_0 \vdash_g f \ \overline{p_i}@r \mid f \ \overline{p'_i}@r \ , \ (\cup_i T_i) \cup [r \mapsto ()] \ , \ \emptyset}$$

$$\frac{T_0 \vdash_g izq \mid izq' \ , \ T \ , \ T'_0 \quad T : (T'_0 \cup T_0) \vdash der \mid der'}{T_0 \vdash_g izq \ der \mid izq' \ der' \ , \ \emptyset \ , \ T'_0}$$

$$\frac{T_0 \vdash_g def_1 \mid def'_1 \ , \ \emptyset \ , \ T_1 \quad T_0 \cup T_1 \vdash_g def_2 \mid def'_2 \ , \ \emptyset \ , \ T_2 \ldots \cup_{i=0}^{n-1} T_i \vdash_g def_n \mid def'_n \ , \ \emptyset \ , \ T_n}{T_0 \vdash_g def_1 \ ; \ \ldots \ ; \ def_n \mid def'_1 \ ; \ \ldots \ ; \ def'_n \ , \ \emptyset \ , \ \cup_{i=0}^{n} T_i}$$

Figura 6.4: Reglas semánticas para definiciones globales

$$\frac{T_0 \not\vdash d \quad UsoDer(d) \quad T'_0 = Rec(T_0, d)}{T_0 \vdash_{dd} d \mid T'_0}$$

Figura 6.5: Reglas semánticas para declaraciones de datos

Capítulo 7

Transformaciones de *desazucaramiento*

En este capítulo se detallan las reglas que se siguen para transformar la sintaxis dulce, con los identificadores ya renombrados y los tipos ya inferidos, a la sintaxis amarga. Para cada posible constructora del árbol abstracto hay una regla para eliminar las construcciones complejas y transformarlas en combinaciones de las construcciones más básicas. En el apéndice D se muestran los detalles de la implementación de cómo se realizan dichas transformaciones. En esta sección, la transformada de una expresión e se denota por $\overline{e}$. En la sección 7.1 pueden verse las reglas más sencillas, mientras que en la sección 7.2 se dedicarán varias subsecciones para explicar en detalle las reglas más complejas.

7.1. Transformaciones simples

A continuación se muestran las transformaciónes más sencillas. En algunos casos se generarán variables frescas. Éstas deben tiparse con los tipos de las expresiones a las que sustituyen, inferidos en la fase de inferencia de tipos *Hindley-Milner*.

$\overline{c} \equiv c$

$\overline{C\ a_1\ \ldots\ a_n\ @\ r} \equiv C\ a_1\ \ldots\ a_n\ @\ r$ **donde** las a_i son constantes o variables

$$\begin{aligned}\overline{C\ e_1\ \ldots\ e_n\ @\ r} \equiv\ & \textbf{let}\ x_1 = \overline{e_1}\ \textbf{in}\\ & \ldots\\ & \textbf{let}\ x_n = \overline{e_n}\ \textbf{in}\\ & C\ x_1\ \ldots\ x_n\ @\ r \quad \textbf{donde las } x_i \text{ representan variables frescas}\end{aligned}$$

$\overline{x} \equiv x$

$\overline{x\ @\ r} \equiv x\ @\ r$

$\overline{x!} \equiv x!$

$\overline{f\ a_1\ \ldots\ a_n\ @\ r} \equiv f\ a_1\ \ldots\ a_n\ @\ r$ **donde** las a_i son constantes o variables

$$\overline{f\ e_1\ \ldots\ e_n\ @\ r} \equiv \begin{array}{l} \textbf{let}\ x_1 = \overline{e_1}\ \textbf{in} \\ \ldots \\ \textbf{let}\ x_n = \overline{e_n}\ \textbf{in} \\ f\ x_1\ \ldots\ x_n\ @\ r \end{array} \quad \textbf{donde}\ x_i \text{ representa variables frescas}$$

Tanto las transformadas de las constructoras como en las de las funciones, en las que ambas tienen expresiones como parámetros, hay que tener en cuenta que sólo se generan variables frescas x_i para expresiones que sean constantes o variables.

$\overline{\textbf{let}\ x_1 = e_1\ \textbf{in}\ e} \equiv \textbf{let}\ x_1 = \overline{e_1}\ \textbf{in}\ \overline{e}$

$\overline{\textbf{let}\ p = e_1\ \textbf{in}\ e} \equiv \overline{\textbf{case}\ e_1\ \textbf{of}\ p \rightarrow e}$

$\overline{\textbf{case}\ x\ \textbf{of}\ alt_1\ ;\ \ldots\ ;\ alt_n} \equiv$ *match (esta función se explica detalladamente en la sección 7.2)*

$\overline{\textbf{case}\ e\ \textbf{of}\ alt_1\ ;\ \ldots\ ;\ alt_n} \equiv \textbf{let}\ x = \overline{e}\ \textbf{in}\ \overline{\textbf{case}\ x\ \textbf{of}\ alt_1\ ;\ \ldots\ ;\ alt_n}$

$$\overline{\textbf{let}\ \{\ def_1\ ;\ \ldots\ ;\ def_n\ \}\ \textbf{in}\ e} \equiv \begin{array}{l} \overline{\textbf{let}\ def_1\ \textbf{in}} \\ \ldots \\ \overline{\textbf{let}\ def_n\ \textbf{in}\ e} \end{array} \quad \begin{array}{l} \textbf{donde no debe existir recursión mutua (en concreto}\ def_i\ \textbf{no puede llamar a}\ def_j\ \textbf{siendo}\ i < j\textbf{)} \end{array}$$

Transformaciones de la parte derecha de las definiciones:

$der\ =\ der_i\ \textbf{where}\ defs$

$\overline{der}\ \equiv\ \overline{der_i\ \textbf{where}\ defs} \equiv \overline{\textbf{let}\ defs\ \textbf{in}\ der_i}$

$der_i\ =\ simple\ \mid\ guardado$

$\overline{simple}\ \equiv\ \overline{e}$

$$\overline{guardado}\ \equiv\ \overline{\mid e_1 = e'_1\ \ldots \mid e_n = e'n} \equiv \begin{array}{l} \overline{\textbf{case}\ e_1\ \textbf{of}} \\ \quad \overline{True \rightarrow e'_1} \\ \quad \overline{False \rightarrow \ldots\ \textbf{case}\ e_n\ \textbf{of}} \\ \qquad \overline{True \rightarrow e'_n} \\ \qquad \overline{False \rightarrow \textbf{error}} \end{array}$$

7.2. Transformaciones complejas

(Basado en [Jon87]).

Las transformaciones más complejas son (1) la que convierte un conjunto ecuaciones con patrones en una sola ecuación y (2) la que convierte un **case** con patrones complejos en una serie de **case** con patrones simples. Afortunadamente, se pueden hacer ambas utilizando una sola función:

$$tr \left(\begin{array}{l} f\ p_{11} \cdots p_{1n} = e_1 \\ \cdots \\ f\ p_{r1} \cdots p_{rn} = e_r \end{array} \right) \quad \stackrel{\text{def}}{=} f\ x_1 \cdots x_n = match \quad [x_1, \ldots, x_n]$$

$$[([p_{11}, \ldots p_{1n}],\ e_1), \ldots, ([p_{r1}, \ldots p_{rn}],\ e_r)]$$
$$error$$

$$tr \left(\begin{array}{l} \textbf{case}\ x\ \textbf{of} \\ \quad p_1 \rightarrow e_1 \\ \quad \cdots \\ \quad p_r \rightarrow e_r \end{array} \right) \quad \stackrel{\text{def}}{=} \quad match \quad [x]$$

$$[([p_1], e_1), \ldots, ([p_r], e_r)]$$
$$error$$

donde:

$match :: [Var] \rightarrow [([Pat], Exp)] \rightarrow Exp \rightarrow Exp$

La idea intuitiva de *match xs qs e* es que *xs* es una lista de variables-argumento de longitud n que hay que encajar en una lista *qs* de ecuaciones y *e* es la expresión por defecto en caso de que todos los encajes fallen. La lista *qs* puede tener cero o más ecuaciones de la forma (ps_i, e_i) donde $ps_i = [p_{i1}, \ldots, p_{in}]$ es una lista de patrones de la misma longitud n que la lista *xs* y e_i es la expresión a ejecutar en caso de que todos los argumentos encajen.

7.2.1. Regla vacía

Cuando no hay que encajar más argumentos, puede suceder que no haya ecuaciones, en cuyo caso se devuelve la expresión por defecto, o que haya al menos una ecuación. Si hay más de una, significa que en la definición original había solape entre los patrones de varias ecuaciones. En ese caso, atendiendo a la semántica de un encaje secuencial, se elige la primera de las ecuaciones que encajan.

$$\begin{array}{lcl} match\ \_\ [\,]\ e & = & e \\ match\ [\,]\ (([\,], e_1) : qs)\ e_2 & = & e_1 \end{array}$$

7.2.2. Regla cuando todos los patrones son una variable

Si el siguiente argumento a encajar se corresponde con una variable en todas las ecuaciones, el encaje tiene éxito y simplemente se sustituye en cada parte derecha la variable-patrón por la variable-argumento:

$$match\ (x : xs)\ [(y_1 : ps_1, e_1), \ldots, (y_r : ps_r, e_r)]\ e = match\ xs\ [(ps_1, e_1[x/y_1]), \ldots, (ps_r, e_r[x/y_r])]\ e$$

7.2.3. Regla cuando todos los patrones son constructoras

Si el siguiente argumento a encajar se corresponde en todas las ecuaciones con un patrón de constructora, entonces la transformación introduce una expresión **case** y se continua haciendo encaje en cada una de sus ramas. Las ecuaciones que presentan constructoras distintas se pueden cambiar de orden dentro de la lista puesto que representan alternativas excluyentes, pero las que empiezan por la misma constructora han de mantener su orden relativo para no alterar la semántica secuencial del encaje. Además, se deben cubrir todas las constructoras declaradas en el tipo de datos.

Sean $C_1, \ldots, C_t$ *todas* las constructoras del tipo del siguiente argumento a encajar. Se agrupan entonces las ecuaciones en t sublistas $qs_1, \ldots, qs_t$, conteniendo cada qs_i todas las ecuaciones que comienzan por la constructora C_i. Cada qs_i es de la forma:

$$qs_i = [(C_i\ p_{11} \cdots p_{1n_i} : ps_{i1},\ e_{i1}), \ldots, (C_i\ p_{s1} \cdots p_{sn_i} : ps_{is},\ e_{is})]$$

Si alguna constructora C_j no está presente, se genera de todos modos una lista qs_j vacía. Para generar un código eficiente, es necesario que cada **case** generado sea completo (i.e. que incluya todas las constructoras) y que mantenga siempre el mismo orden (i.e. alfabético) $C_1, \ldots, C_t$ en las alternativas:

$$\begin{array}{lll} match\ (x : xs) & (qs_1 +\!\!+ \cdots +\!\!+ qs_t)\ e & = \\ & \textbf{case}\ x\ \textbf{of} & \\ & \quad C_1\ x_{11} \cdots x_{1n_1} & \rightarrow match\ (x_{11} : \cdots : x_{1n_1} : xs)\ qs'_1\ e \\ & \quad \cdots & \\ & \quad C_t\ x_{t1} \cdots x_{tn_t} & \rightarrow match\ (x_{t1} : \cdots : x_{tn_t} : xs)\ qs'_t\ e \end{array}$$

donde las qs'_i son de la forma:

$$qs'_i = [(p_{11} : \cdots : p_{1n_i} : ps_{i1},\ e_{i1}), \ldots, (p_{s1} : \cdots : p_{sn_i} : ps_{is},\ e_{is})]$$

En el caso de que estas constructoras sean constantes literales enteras, al ser los números enteros infinitos se añade una variable _, como caso en el que encajan todos los demás no recogidos y asignándole la lista vacía a la qs'_i asociada.

7.2.4. Regla mixta

Puede ocurrir que el siguiente argumento a encajar se encuentre con una situación mixta en la que algunas ecuaciones empiezan por constructora y otras empiezan por variable. Entonces hay que partir la lista qs de ecuaciones en fragmentos uniformes $qs_1, \ldots, qs_s$ de forma que dentro de cada fragmento qs_i todas las ecuaciones empiecen por variable o todas empiecen por constructora. Además se ha de cumplir $qs = qs_1 +\!\!+ \cdots +\!\!+ qs_s$. Entonces, la regla utiliza la expresión por defecto para realizar un encaje secuencial en el que es posible que el argumento sufra más de un encaje consecutivo:

$$\begin{array}{lll} match\ (x : xs)\ (qs_1 +\!\!+ \cdots +\!\!+ qs_s)\ e = & match\ (x : xs)\ qs_1\ (& \\ & \quad match\ (x : xs)\ qs_2\ (& \\ & \quad\quad \cdots (match\ (x : xs)\ qs_s\ e) \cdots)) & \end{array}$$

7.2.5. Ejemplo

Sea la definición:

```
zipWith f []     ys     = []
zipWith f xs     []     = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Utilizando las reglas anteriores el resultado obtenido es:

```
zipWith g zs ws = case zs of
                    []   -> []
                    z:zz -> case ws of
                              []   -> []
                              w:ww -> case zs of
                                        []   -> error
                                        r:rr -> case ws of
                                                  []   -> error
                                                  t:tt -> g r t : zipWith g rr tt
```

Capítulo 8

Ejemplo completo

En este capítulo se ilustra el proceso llevado a cabo en cada fase de compilación mediante un ejemplo. Se mostrará paso a paso el *pretty printing* del árbol abstracto resultante de cada fase. Este ejemplo y el resto de las pruebas se han realizado con un programa principal implementado en el módulo `Main.hs`, encargado de leer un archivo de entrada e invocar a las fases correspondientes, para imprimirlo posteriormente de una forma amigable, similar a cómo estaría escrito el programa en SAFE, añadiéndole la información necesaria en cada caso.

En primer lugar se muestra el archivo de entrada original. Se trata de la función `mergesort` escrita en Haskell con las cualidades propias de SAFE. Recuérdese que los identificadores precedidos por el símbolo @ son las regiones donde se construirán las listas, y los símbolos ! indican que las listas de entrada se destruirán tras su lectura, por ello en la llamada recursiva se vuelven a construir.

$$
\begin{aligned}
&mergeD :: [Int]!@\rho \to [Int]!@\rho \to \rho \to [Int]@\rho \\
&mergeD\ [\,]!\ ys!\ @r = ys! \\
&mergeD\ (x : xs)!\ [\,]!\ @r = (x : xs!)\ @r \\
&mergeD\ (x : xs)!\ (y : ys)!\ @r \\
&\quad \mid x \leq y = \ (x : mergeD\ xs\ ((y : ys!)\ @r)\ @r)\ @r \\
&\quad \mid x > y = \ (y : mergeD\ ((x : xs!)\ @r)\ ys@r)\ @r
\end{aligned}
$$

Tras el análisis sintáctico parece que no hay ningún cambio aparente, ya que lo único que hace es comprobar la corrección sintáctica del programa y construir el árbol abstracto, que al ser impreso amigablemente se parece mucho a la versión original. Únicamente destacar que las constructoras infijas son impresas como las prefijas, ya que la sintaxis abstracta no diferencia entre estos casos. Como consecuencia, el módulo de impresión amigable lo imprime de este modo.

$$
\begin{aligned}
&mergeD :: [Int]!@\rho \to [Int]!@\rho \to \rho \to [Int]@\rho \\
&mergeD\ [\,]!\ ys!\ @r = ys! \\
&mergeD\ (:\ x\ xs)!\ [\,]!\ @r =: x\ xs!@r \\
&mergeD\ (:\ x\ xs)!\ (:\ y\ ys)!@r \\
&\quad \mid\ \leq\ x\ y = \ :\ x\ (mergeD\ xs\ (:\ y\ ys!@r)\ @r)\ @r \\
&\quad \mid\ >\ x\ y = \ :\ y\ (mergeD\ (:\ x\ xs!@r)\ ys@r)\ @r
\end{aligned}
$$

El análisis semántico sí que muestra resultados más visibles, tras seguir las reglas y comprobar que el programa es semánticamente correcto los cambios apreciables son el renombramiento de las variables que, como se ve en el ejemplo, se mantiene correcto para cada ámbito.

$$
\begin{array}{l}
mergeD :: [Int]!@\rho \to [Int]!@\rho \to \rho \to [Int]@\rho \\
mergeD\ [\,]!\ 17ys!\ @r = 17ys! \\
mergeD\ (:\ 18x\ 19xs)!\ [\,]!\ @r =: 18x\ 19xs!@r \\
mergeD\ (:\ 20x\ 21xs)!\ (:\ 22y\ 23ys)!@r \\
\quad |\ \ \leq 20x\ 22y = \ :\ 20x\ (mergeD\ 21xs\ (:\ 22y\ 23ys!@r)\ @r)\ @r \\
\quad |\ \ > 20x\ 22y = \ :\ 22y\ (mergeD\ (:\ 20x\ 21xs!@r)\ 23ys@r)\ @r
\end{array}
$$

En el siguiente resultado se ha utilizado otra versión de *pretty printer* que imprime los tipos de cada una de las variables y funciones, ya que en este paso se ha aplicado una modificación del algoritmo de inferencia de tipos Hindley-Milner para inferir los tipos de cada uno de ellos. En este caso, al haber sido declarado el tipo de la función, se comprueba que deben ser compatibles —el mismo tipo u otro más general—.

$$
\begin{array}{l}
mergeD :: [Int]!@\rho \to [Int]!@\rho \to \rho \to [Int]@\rho \\
mergeD\ [\,]\{[Int]@a10\}!\ 17ys\{[Int]@a10\}!\ @r \\
\quad =\ 17ys!\{[Int]@a10\} \\
\\
mergeD\ (:\ 18x\{Int\}\ 19xs\{[Int]@a10\})\{[Int]@a10\}!\ [\,]\{[Int]@a10\}!\ @r \\
\quad =\ :\ 18x\{Int\}\ 19xs!\{[Int]@a10\}@r\{[Int]@a10\} \\
\\
mergeD\ (:\ 20x\{Int\}\ 21xs\{[Int]@a10\})\{[Int]@a10\}! \\
\qquad (:\ 22y\{Int\}\ 23ys\{[Int]@a10\})\{[Int]@a10\}!\ @r \\
\quad |\ \leq\ 20x\{Int\}\ 22y\{Int\}\ \{Bool\} \\
\quad =\ :\ 20x\{Int\}\ (mergeD\ 21xs\{[Int]@a10\}\ (:\ 22y\{Int\}\ 23ys!\{[Int]@a10\}@r) \\
\qquad \{[Int]@a10\}@r)\ \{[Int]@a10\}@r\ \{[Int]@a10\} \\
\\
\quad |\ >\ 20x\{Int\}\ 22y\{Int\}\ \{Bool\} \\
\quad =\ :\ 22y\{Int\}\ (mergeD\ (:\ 20x\{Int\}\ 21xs!\{[Int]@a10\}@r)\ \{[Int]@a10\}\ 23ys\{[Int]@a10\}@r) \\
\qquad \{[Int]@a10\}\ @r\ \{[Int]@a10\}
\end{array}
$$

La siguiente figura es visiblemente más complicada que el ejemplo inicial. Esto se debe a que la sintaxis *amarga* de SAFE es mucho menos amigable para el programador, de ahí la utilización de una sintaxis dulce y estas transformaciones *amargativas* que han sido aplicadas a este resultado. Se puede ver que ahora hay una única regla, y el encaje de patrones para saber que regla debería actuar en cada caso ha sido sustituido por las dos primeras sentencias **case!** anidadas que hacen que encaje según los patrones de las reglas. Una nota a destacar es que los patrones *16_x* y *17_x* han dejado de estar marcados como destructivos ya que lo que en realidad se encarga de destruir esa memoria son esas sentencias **case!**. Las dos sentencias **case** siguientes corresponden a la regla de transformar las guardas, véase que la última alternativa *False* da como resultado un error, derivado de que no han sido contemplados todos los casos en las guardas —aunque este no es el caso— y las cláusulas **let** corresponden a la transformación de expresiones complejas en variables simples. Por último, se ve que todas las variables utilizadas son frescas. Este es un efecto lateral derivado de la función `match` explicada en la sección 7.2.

```
mergeD :: [Int]!@ρ → [Int]!@ρ → ρ → [Int]@ρ
mergeD 16_x 17_x @r =
  case! 16_x of
    (: 27_x 28_x) → case! 17_x of
                      (: 29_x 30_x) → let 19_x = ≤ 27_x 29_x
                                      in case 19_x of
                                           False → let 23_x = > 27_x 29_x
                                                   in case 23_x of
                                                        False → error guardas
                                                        True → let 24_x = let 25_x = let 26_x = 28_x!
                                                                          in : 27_x 26_x @r
                                                               in mergeD 25_x 30_x @r
                                                  in : 29_x 24_x @r
                                           True → let 20_x = let 21_x = let 22_x = 30_x!
                                                             in : 29_x 22_x @r
                                                  in mergeD 28_x 21_x @r
                                      in : 27_x 20_x @r
                      [ ] → let 18_x = 28_x!
                            in : 27_x 18_x @r
    [ ] → 17_x!
```

Capítulo 9

Conclusiones

La contribución fundamental de este proyecto ha sido la construcción de una herramienta práctica capaz de ejecutar cada una de las fases descritas en este trabajo, permitiendo visualizar los resultados en un fichero de texto sobre el que se ha realizado un proceso de *pretty-printing*. De este modo ha sido posible conocer si la idea de desarrollar un compilador funcional con gestión explícita de memoria es factible y práctica, gracias a las pruebas de varios ejemplos de interés.

Durante el desarrollo del mismo se han utilizado las siguientes herramientas y tecnologías:

- La herramienta *Alex* para generar el analizador léxico.
- La herramienta *Happy* para generar el analizador sintáctico.
- El compilador *GHC* de Haskell junto con una variedad de librerías:
  - La librería `PPrint` para visualizar los resultados del compilador en ficheros de texto plano.
  - Las librerías `IO` y `System` para realizar entrada-salida utilizando *handles*.
  - La librería `Data` para utilizar los tipos de datos abstractos `Map` y `List` en el analizador semántico y en el módulo encargado de las transformaciones *amargativas*.
- El lenguaje *LaTeX* para desarrollar la presente memoria.

Y el resultado de la implementación, que ha sido organizada en diferentes módulos, ha sido el siguiente:

- El módulo `Alexer.x` que implementa el analizador léxico.
- El módulo `HParser.y` que implementa el analizador sintáctico.
- El módulo `SParser.hs` que implementa el analizador semántico.
- El módulo `Transformer.hs` encargado de las transformaciones amargativas.
- El módulo `Printer.hs` que realiza el proceso de *pretty-printing* al resultado obtenido por el compilador.
- El módulo `Main.hs` que constituye el punto de entrada de la herramienta.

También se han utilizado los siguientes módulos, desarrollados por colaboradores del proyecto e imprescindibles para el compilador:

- `SintaxisAbstracta.hs`, que contiene la definición del árbol abstracto.
- `SintaxisAbstractaUtil.hs`, que contiene funciones de utilidad para tratar con el árbol abstracto.
- `HMInference.hs`, módulo encargado de realizar la inferencia de tipos *Hindley-Milner*.
- `PPrint.hs`, módulo basado en el *prettier printer* de Philip Wadlers.
- `Preludio.hs`, fichero que incluye definiciones de los operadores básicos del lenguaje.
- `Substitution.hs`, módulo donde se define el tipo abstracto de datos Subst, utilizado para transformar variables de tipo de expresiones.

Además, el grupo que conforma el presente trabajo ha adquirido conocimientos avanzados de programacion funcional a lo largo del desarrollo del mismo: durante nuestros comienzos, a menudo implementábamos las funciones recursivamente, dificultando la comprensión del código, pero a medida que progresábamos fuimos aprovechándonos de la potencia de las funciones de orden superior, que además simplificaban en gran medida la implementación de las funciones.

Como trabajo futuro se propone extender el lenguaje funcional SAFE para soportar funciones de orden superior, y enriquecer las decoraciones para que incluyan información de tamaño y consumo de memoria, con el objetivo de comprobar/inferir cotas superiores para las regiones.

Apéndice A

Código del análisis léxico

Apéndice B

Código del análisis sintáctico

Apéndice C

Código del análisis semántico

Apéndice D

Código de las transformaciones

Bibliografía

[Jon87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[PS04] R. Peña and C. Segura. A first-order functional language for reasoning about heap consumption. In *Proceedings of the 16th International Workshop on Implementation of Functional Languages, IFL'04. Technical Report 0408, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel*, pages 64–80, 2004.

[PSM06] R. Peña, C. Segura, and M. Montenegro. A sharing analysis for safe. In *Proceedings of the Seventh Symposium on Trends in Functional Programming, TFP'06*, pages 205–221, 2006.